

APPLICATION OF THE FINITE ELEMENT METHOD TO SOME SIMPLE SYSTEMS IN  
ONE AND TWO DIMENSIONS

Jason C. Hunnell, B.S.

Problem in Lieu of Thesis Prepared for the Degree of  
MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2002

APPROVED:

Sandra J. Ward, Major Professor  
William D. Deering, Committee Member  
Carlos A. Ordonez, Committee Member  
Samuel E. Matteson, Chair of the Department of  
Physics  
C. Neal Tate, Dean of the Robert B. Toulouse  
School of Graduate Studies

Hunnell, Jason C., Application of the finite element method to some simple systems in one and two dimensions. Master of Science (Physics), May 2002, 83 pp., 10 tables, 16 illustrations, references, 6 titles.

The finite element method (FEM) is reviewed and applied to the one-dimensional eigensystems of the isotropic harmonic oscillator, finite well, infinite well and radial hydrogen atom, and the two-dimensional eigensystems of the isotropic harmonic oscillator and the propagational modes of sound in a rectangular cavity. Computer codes that I developed were introduced and utilized to find accurate results for the FEM eigensolutions. One of the computer codes was modified and applied to the one-dimensional unbound quantum mechanical system of a square barrier potential and also provided accurate results.

## TABLE OF CONTENTS

	Page
LIST OF TABLES .....	iv
LIST OF FIGURES .....	v
I. INTRODUCTION .....	1
II. FINITE ELEMENT REVIEW .....	3
A. The General Eigenproblem.....	3
B. Element Equations .....	3
1. The element concept .....	3
2. Shape functions.....	6
a. A simple element .....	6
b. Higher order elements .....	8
c. Higher order nodes.....	11
3. Numerical integration.....	13
C. Assembly .....	14
III. THE ONE-DIMENSIONAL CODE .....	18
IV. BOUND SYSTEMS IN ONE DIMENSION.....	21
A. Harmonic Oscillator.....	21
B. Particle in a Finite Well.....	27
C. Particle in an Infinite Well .....	28

D. Hydrogen Atom Energy Levels.....	29
V. UNBOUND SYSTEMS IN ONE DIMENSION .....	31
A. Changes to the One-Dimensional Code .....	32
B. Tunneling Through a Barrier.....	32
VI. A SIMPLE TWO-DIMENSIONAL FINITE ELEMENT METHOD .....	34
A. The Two-Dimensional Technique .....	35
B. Two-Dimensional Applications.....	40
1. Isotropic harmonic oscillator.....	40
2. Propagation of sound in a rectangular cavity .....	42
VII. CONCLUSION .....	45
APPENDIX.....	46
DATA INPUT INSTRUCTIONS AND LISTINGS FOR TISEFEA AND FEM2DLINEAR .....	46
A. Data Input Instructions for TISEFEA .....	47
B. TISEFEA Code Listing .....	49
C. FEM2DLinear Code Listing.....	68
REFERENCES.....	83

## LIST OF TABLES

	Page
Table 1 – Public interfaces.....	19
Table 2 - Harmonic oscillator eigenvalues for some elements of various type with domain from $-9$ to $9$ .....	26
Table 3 – Particle in a finite well of barrier width $6$ with domain from $-10$ to $10$ . There are 100 elements of type 2-3. ....	28
Table 4 – Particle in an approximated infinite well of barrier height $1E10$ , barrier width $6$ , and domain from $-4$ to $4$ . There are 40 elements of type 3-1. ....	29
Table 5 – Electron in the coulomb potential of the hydrogen atom. The solution domain is from $0$ to $100$ and there are 100 elements of type 5-1.....	30
Table 6 – Tunneling through a square barrier of height $4$ . All elements are of type 2-2. ....	33
Table 7 – Estimated eigenvalues, $E_n$ , for the two-dimensional isotropic harmonic oscillator using linear triangular elements with $K$ linear divisions over the domain $(-6 < x < 6, -6 < y < 6)$ . ....	40
Table 8 - Estimated eigenfunctions for the two-dimensional isotropic harmonic oscillator using linear triangular elements with 50 linear divisions over the domain $(-6 < x < 6, -6 < y < 6)$ .....	41
Table 9 - Estimated cut-off frequencies, $f_n$ , for the propagation of sound in a rectangular cavity using linear triangular elements with $K$ linear divisions over $(-12 < x < 12, -14 < y < 14)$ .....	43
Table 10 - Estimated eigenfunctions for the propagation of sound in a rectangular cavity using linear triangular elements with 50 linear divisions over $(-12 < x < 12, -14 < y < 14)$ . ....	44

## LIST OF FIGURES

	Page
Figure 1 – A linear element. ....	6
Figure 2 – Linear isoparametric element.....	7
Figure 3 – Quadratic isoparametric element.....	9
Figure 4 – An element with two nodes and two degrees of freedom per node.....	11
Figure 5 – TISEFEA object model.....	19
Figure 6 – Harmonic oscillator eigenfunctions with 10 elements of type 2-1 with a domain from $-3$ to $3$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	22
Figure 7 - Harmonic oscillator eigenfunctions with 20 elements of type 2-1 with a domain from $-3$ to $3$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	22
Figure 8 - Harmonic oscillator eigenfunctions with 40 elements of type 2-1 with a domain from $-6$ to $6$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	23
Figure 9 - Harmonic oscillator eigenfunctions with 80 elements of type 2-1 with a domain from $-6$ to $6$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	24
Figure 10 - Harmonic oscillator eigenfunctions with 160 elements of type 2-1 with a domain from $-6$ to $6$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	24
Figure 11 - Harmonic oscillator eigenfunctions with 40 elements of type 3-1 with a domain from $-6$ to $6$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	25
Figure 12 - Harmonic oscillator eigenfunctions with 40 elements of type 4-1 with a domain from $-6$ to $6$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	25

Figure 13 - Harmonic oscillator eigenfunctions with 40 elements of type 5-1 with a domain from $-6$ to $6$ . The legend shows the corresponding eigenvalue for each eigenfunction. ....	25
Figure 14 – A linear triangular element.....	36
Figure 15 – Shape function for a linear triangular element.....	37
Figure 16 – Two neighboring linear triangular elements .....	39

## I. INTRODUCTION

The finite element method (FEM) has long been a tool utilized by the engineering profession, and has only more recently come into increased use by the physics community.<sup>1,2,3,4</sup> Historically the FEM evolved out of complex systems solutions in structural mechanics<sup>1</sup> and from there permeated other scientific areas including fluid dynamics, heat transfer, acoustics, and electromagnetism.<sup>1,2</sup> More recently the FEM has been applied to the areas of quantum mechanics and solid state physics.<sup>3,4</sup> Burnett<sup>2</sup> offers the following introductory definition of the FEM:

The FEM is a computer-aided mathematical technique for obtaining approximate numerical solutions to the abstract equations of calculus that predict the response of physical systems subjected to external influences.

The basic idea is to break up the domain of a problem into smaller elements unto which the “laws” of the larger domain still apply. These smaller elements are mathematically and computationally coupled together so that the numerical solution of all these smaller elements approximates the solution of the larger, more complex domain. In this way problems with complicated domains and boundary conditions can be tackled that may not have been solvable by other analytical or numerical techniques.

In this project, I have first applied the FEM to forms of the one-dimensional, time-independent Schrödinger equation,

$$-\frac{\hbar^2}{2m}\Psi''(x) + V(x)\Psi(x) = E\Psi(x). \quad (1)$$

Here Eq. (1) is first considered as a general eigenproblem, where both the eigenvalues,  $E$ , and the eigenfunctions,  $\Psi(x)$ , are unknowns, and I review the FEM as applied to this



general eigenproblem. I then introduce the object-oriented, C++, computer code I developed to help solve this general eigenproblem. Using the computer code, one-dimensional eigensolutions to the isotropic harmonic oscillator, the particle in an infinite well, the particle in a finite well, and radial solutions to the hydrogen atom are all accurately approximated.

Eq. (1) is then considered in its standard form, where the energy,  $E$ , is now known, and the problems are no longer eigenproblems, but simple systems of linear equations. A system of this type is the problem of quantum mechanical tunneling through a square barrier. I modified the FEM code and applied it to this system in a technique similar to that used by Goloskie, et al.<sup>4</sup> This new code is shown to approximate the reflection and transmission coefficients of square barrier potentials.

Finally, I investigate the FEM applied to some eigensystems in two dimensions, in particular, the isotropic two-dimensional harmonic oscillator and the propagation of sound in a rectangular cavity. The background, computational technique, computer code, and resulting approximations are all reported.

## II. FINITE ELEMENT REVIEW

### A. THE GENERAL EIGENPROBLEM

Consider the following one-dimensional eigenproblem (the mathematical notation used throughout is similar to that used by Burnett<sup>2</sup>),

$$-\alpha(x)\Psi''(x) + \beta(x)\Psi'(x) - \lambda\gamma(x)\Psi(x) = 0, \quad (2)$$

where  $\alpha$ ,  $\beta$ , and  $\gamma$  are all quadratic polynomials in  $x$ , and  $\lambda$  is an unknown scalar. The domain is any interval  $a < x < b$ , and the boundary conditions are assigned from the following,

$$\Psi(x = a) = 0 \quad \text{or} \quad \Psi'(x = a) = 0, \quad (3a)$$

and

$$\Psi(x = b) = 0 \quad \text{or} \quad \Psi'(x = b) = 0. \quad (3b)$$

The eigenvalues,  $\lambda$ , and the eigenvectors,  $\Psi(x)$ , are sought.

### B. ELEMENT EQUATIONS

#### 1. The element concept

The domain must first be broken into smaller pieces called elements (not necessarily all of equal length). The solution will be approximated within each element by a superposition of local basis functions. The form of the local basis functions will be discussed shortly. The approximate solution for the  $e^{\text{th}}$  element is then,

$$\Psi^{(e)}(x) = \sum_{j=1}^m a_j \varphi_j^{(e)}(x), \quad (4)$$

where the  $a_j$  are coefficients, and the  $\phi_j^{(e)}$  are the local basis functions. Thus, the approximate solution over the entire domain is

$$\Psi(x) = \sum_{e=1}^n \Psi^{(e)}(x) = \sum_{e=1}^n \sum_{j=1}^m a_j \phi_j^{(e)}(x). \quad (5)$$

A number of linear equations will need to be solved to find the unknown coefficients. To arrive at this system of linear equations the Galerkin method of weighted residuals is utilized<sup>1,2</sup>. In the Galerkin method, the expression in Eq. (2) is multiplied by the local basis functions and then integrated over the local element domain. The following system for the  $e^{\text{th}}$  element is thus arrived at,

$$\int^{(e)} [-\alpha(x) \Psi^{(e)''}(x) + \beta(x) \Psi^{(e)}(x) - \lambda \gamma(x) \Psi^{(e)}(x)] \phi_i^{(e)}(x) dx = 0, \quad (6)$$

where  $i$  ranges from 1 to  $m$ .

In order to eliminate the second-order derivative, and to introduce boundary terms, the second-order term is integrated by parts. This leads to

$$\begin{aligned} \int^{(e)} [\alpha(x) \phi_i^{(e)'}(x) \Psi^{(e)'}(x)] dx + \int^{(e)} [\beta(x) \phi_i^{(e)}(x) \Psi^{(e)}(x)] dx \\ - \lambda \int^{(e)} [\gamma(x) \phi_i^{(e)}(x) \Psi^{(e)}(x)] dx = [\alpha(x) \phi_i^{(e)}(x) \Psi^{(e)'}(x)]_{x_a^{(e)}}^{x_b^{(e)}} \end{aligned} \quad (7)$$

where  $x_a^{(e)}$  and  $x_b^{(e)}$  are the boundaries for the  $e^{\text{th}}$  element. For the eigenproblem, these boundary terms must vanish for each element. Two properties lead to this conclusion. First, the elements located on the boundaries of the domain must adhere to the domain boundary conditions, Eqs. (3a) and (3b), which force the function or its derivative to zero, thus eliminating those boundary terms. Second, when the element equations are coupled together, the boundary terms common between elements will disappear, as the function

and its derivative are continuous across element boundaries. This then results in the following

$$\int^{(e)} [\alpha(x) \varphi_i^{(e)'}(x) \Psi^{(e)'}(x)] dx + \int^{(e)} [\beta(x) \varphi_i^{(e)}(x) \Psi^{(e)}(x)] dx - \lambda \int^{(e)} [\gamma(x) \varphi_i^{(e)}(x) \Psi^{(e)}(x)] dx = 0. \quad (8)$$

Now substitute the approximate solution for the  $e^{\text{th}}$  element, Eq. (4), into Eq. (8), leading to element equations of the form

$$\sum_{j=1}^m \left[ \int^{(e)} [\alpha(x) \varphi_i^{(e)'}(x) \varphi_j^{(e)'}(x)] dx + \int^{(e)} [\beta(x) \varphi_i^{(e)}(x) \varphi_j^{(e)}(x)] dx - \lambda \int^{(e)} [\gamma(x) \varphi_i^{(e)}(x) \varphi_j^{(e)}(x)] dx \right] a_j = 0. \quad (9)$$

Finally, the element equations in Eq. (9) are written in the matrix form

$$[K]^{(e)} \{a\} - \lambda [M]^{(e)} \{a\} = \{0\}, \quad (10a)$$

where

$$[K]^{(e)} = [K\alpha]^{(e)} + [K\beta]^{(e)}, \quad (10b)$$

and

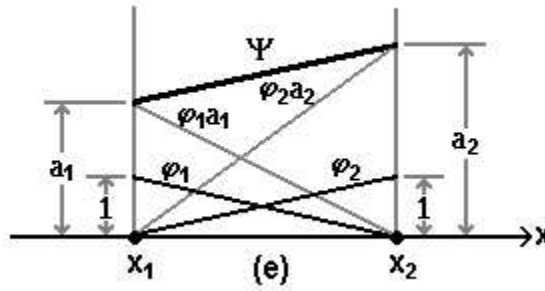
$$\begin{aligned} K\alpha_{ij}^{(e)} &= \int^{(e)} [\alpha(x) \varphi_i^{(e)'}(x) \varphi_j^{(e)'}(x)] dx, \\ K\beta_{ij}^{(e)} &= \int^{(e)} [\beta(x) \varphi_i^{(e)}(x) \varphi_j^{(e)}(x)] dx, \\ M_{ij}^{(e)} &= \int^{(e)} [\gamma(x) \varphi_i^{(e)}(x) \varphi_j^{(e)}(x)] dx. \end{aligned} \quad (11)$$

A number of integrations are required to fill the system of element equations. Typically the matrix  $[K]$  is referred to as the stiffness matrix, while the matrix  $[M]$  is referred to as the mass matrix.

## 2. Shape functions

To proceed with the derivation of the element equations, specific expressions for the local basis functions, or shape functions as they are commonly referred, need to be derived.

**Figure 1 – A linear element.**



a. A simple element

Consider the element, (e), in Figure 1, above. The approximate solution in this linear element is a linear combination of two shape functions,

$$\Psi^{(e)}(x) = a_1 \phi_1^{(e)}(x) + a_2 \phi_2^{(e)}(x). \quad (12)$$

The shape functions must adhere to an interpolation property

$$\phi_i^{(e)}(x_j) = \delta_{ij}, \quad (13)$$

where  $\delta_{ij}$  is the Kronecker delta. Notice that (e) has a left bound of  $x_1$  and a right bound of  $x_2$ , and the shape functions are mapped over these global coordinates. Also note that

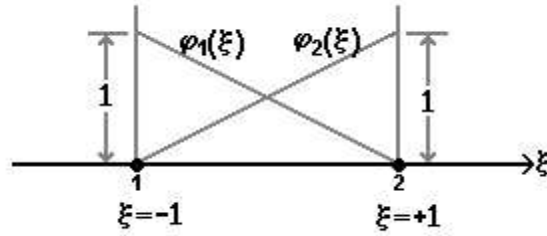
$$\Psi^{(e)}(x_i) = a_i. \quad (14)$$

While the specific form of the shape functions is easily determined,

$$\begin{aligned} \phi_1^{(e)}(x) &= \frac{x_2 - x}{x_2 - x_1}, \\ \phi_2^{(e)}(x) &= \frac{x - x_1}{x_2 - x_1}, \end{aligned} \quad (15)$$

these direct shape functions mapped over global coordinates are not particularly well suited for the numerical computations that must be further made on them. Shape functions that are better suited for the numerical computations that will follow are those from an isoparametric type of element.

**Figure 2 – Linear isoparametric element.**



Isoparametric elements are typified by their use of one parent element with a set of local coordinates. This parent element is then transformed into the global domain coordinates for each of the individual elements. In Figure 2 there is illustrated an isoparametric parent element with its own local coordinate system,  $\xi$ , where  $\xi$  ranges from  $-1$  to  $+1$ . The parent shape functions, while taking on the same form as Eqs. (15), are now functions of  $\xi$ . Notice that for each element there is the mapping

$$\begin{aligned}\xi = -1 &\rightarrow x = x_1^{(e)} \\ \xi = +1 &\rightarrow x = x_2^{(e)} .\end{aligned}\tag{16}$$

The parent shape functions then adopt the following form,

$$\begin{aligned}\phi_1(\xi) &= \frac{1}{2}(1 - \xi) \\ \phi_2(\xi) &= \frac{1}{2}(1 + \xi).\end{aligned}\tag{17}$$

Transformations from the local coordinates,  $\xi$ , to the global coordinates,  $x$ , can be made for each element (e), with the mapping,

$$x = \chi^{(e)}(\xi) = \sum_{k=1}^2 x_k^{(e)} \varphi_k(\xi) x_1^{(e)} = \frac{1}{2}(1 - \xi) + x_2^{(e)} \frac{1}{2}(1 + \xi). \quad (18)$$

Furthermore, the transformation of the shape function derivatives, from global to local parent, become

$$\frac{d\varphi_i^{(e)}(x)}{dx} = \frac{d\varphi_i^{(e)}(\chi^{(e)}(x))}{d\xi} \frac{d\xi}{dx} = \frac{1}{J^{(e)}(\xi)} \frac{d\varphi_i(\xi)}{d\xi}, \quad (19)$$

where  $J^{(e)}(\xi)$  is the Jacobian. For all isoparametric elements derived here,

$$J^{(e)}(\xi) = \frac{dx}{d\xi} = \frac{x_R^{(e)} - x_L^{(e)}}{2}, \quad (20)$$

where  $x_R^{(e)}$  and  $x_L^{(e)}$  are the right and left bounds of the element (e).

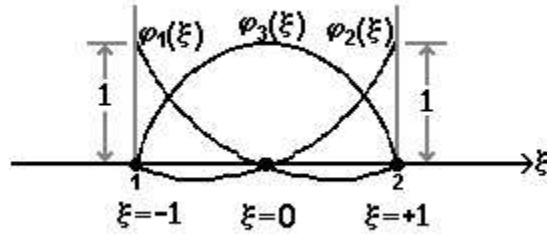
Finally, for isoparametric elements, noting that  $dx = J^{(e)}(\xi)d\xi$ , the element equations, Eqs. (11), become

$$\begin{aligned} K\alpha_{ij}^{(e)} &= \int_{-1}^{+1} \alpha(\chi^{(e)}(\xi)) \frac{1}{J^{(e)}(\xi)} \varphi_i'(\xi) \frac{1}{J^{(e)}(\xi)} \varphi_j'(\xi) J^{(e)}(\xi) d\xi, \\ K\beta_{ij}^{(e)} &= \int_{-1}^{+1} \beta(\chi^{(e)}(\xi)) \varphi_i(\xi) \varphi_j(\xi) J^{(e)}(\xi) d\xi, \\ M_{ij}^{(e)} &= \int_{-1}^{+1} \gamma(\chi^{(e)}(\xi)) \varphi_i(\xi) \varphi_j(\xi) J^{(e)}(\xi) d\xi. \end{aligned} \quad (21)$$

### b. Higher order elements

Linear elements can be characterized as containing two nodes, one at either endpoint, but elements are not restricted to only two nodes. Elements of a higher order than linear can be formulated by increasing the number of nodes within each element. For instance, a quadratic element may be formulated that contains three nodes, one at either endpoint, and one in the center of the element. Such an element is illustrated in Figure 3.

Figure 3 – Quadratic isoparametric element.



Similar to a linear element, the approximate solution within this element is a linear combination of shape functions. A quadratic element contains three nodes, thus the approximate solution is a linear combination of the three shape functions,

$$\Psi^{(e)}(x) = a_1 \varphi_1^{(e)}(x) + a_2 \varphi_2^{(e)}(x) + a_3 \varphi_3^{(e)}(x). \quad (22)$$

Note that these shape functions must adhere to the interpolation property, Eq. (13), and that

$$\Psi^{(e)}(x_i) = a_i. \quad (23)$$

Each shape function takes on the form of a quadratic polynomial in  $x$ , for instance the first shape function is,

$$\varphi_1^{(e)}(x) = b_1 + b_2 x + b_3 x^2, \quad (24)$$

where the  $b_i$  are constants. These constants may be found by applying the interpolation property, Eq. (13), to each of the three nodal points,

$$\begin{aligned} b_1 + b_2 x_1 + b_3 x_1^2 &= 1, \\ b_1 + b_2 x_2 + b_3 x_2^2 &= 0, \\ b_1 + b_2 x_3 + b_3 x_3^2 &= 0, \end{aligned} \quad (25)$$

then solving for the  $b_i$  in terms of the  $x_i$ , and finally substituting back into Eq. (24). The first shape function is thus determined to be



$$\varphi_1^{(e)}(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}, \quad (26a)$$

while the other shape functions can similarly be found to be

$$\begin{aligned} \varphi_2^{(e)}(x) &= \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)}, \\ \varphi_3^{(e)}(x) &= \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}. \end{aligned} \quad (26b)$$

It should be noted that the quadratic shape functions, Eqs. (26), and the linear shape functions, Eqs. (15), are from a family of shape functions known as the Lagrangian interpolation polynomials, and knowing this, Lagrangian interpolation polynomials of any higher degree (more nodes) can easily be derived.

Like the linear shape functions, the quadratic shape functions can be cast into an isoparametric form by applying a mapping,

$$\begin{aligned} \xi = -1 &\rightarrow x = x_1^{(e)}, \\ \xi = 0 &\rightarrow x = x_2^{(e)}, \\ \xi = +1 &\rightarrow x = x_3^{(e)}. \end{aligned} \quad (27)$$

Their isoparametric form is

$$\begin{aligned} \varphi_1^{(e)}(\xi) &= \frac{1}{2}\xi(\xi - 1), \\ \varphi_2^{(e)}(\xi) &= (\xi + 1)(\xi - 1), \\ \varphi_3^{(e)}(\xi) &= \frac{1}{2}\xi(\xi + 1), \end{aligned} \quad (28)$$

and the transformation from  $\xi$  to  $x$  is made with

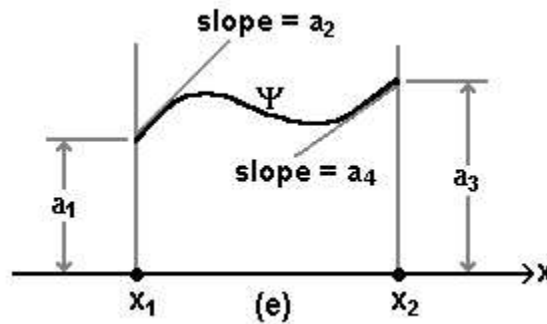
$$\begin{aligned} x = \chi^{(e)}(\xi) &= \sum_{k=1}^3 x_k^{(e)} \varphi_k^{(e)}(\xi) \\ &= x_1^{(e)} \frac{1}{2}\xi(\xi - 1) + x_2^{(e)}(\xi + 1)(\xi - 1) + x_3^{(e)} \frac{1}{2}\xi(\xi + 1). \end{aligned} \quad (29)$$

Using Eqs. (28) and (29) in Eqs. (21), the element equations for isoparametric quadratic elements can be found. Finally, this technique can clearly be used to derive element equations for any higher order elements.

### c. Higher order nodes

The elements investigated so far (based on Lagrangian interpolation polynomials) have only demanded interpolation of the approximate solution,  $\Psi(x)$ . While this interpolation demands that  $\Psi(x)$  be continuous across element boundaries, it does not demand continuity in any of the derivatives of  $\Psi(x)$ . That is, the previously derived elements can not guarantee the continuity of  $\Psi'(x)$ , or any higher derivatives, between elements. If such continuity is desirable, the *order* or *degree of freedom* of each node needs to be increased.

**Figure 4 – An element with two nodes and two degrees of freedom per node.**



Referring to Figure 4, element (e) contains two nodes with two degrees of freedom each. This element interpolates the approximate solution,  $\Psi(x)$ , and its first derivative,  $\Psi'(x)$ , and will guarantee the continuity of both across element boundaries. The approximate solution within this element is the familiar linear combination of shape functions,

$$\Psi^{(e)}(x) = a_1 \phi_1^{(e)}(x) + a_2 \phi_2^{(e)}(x) + a_3 \phi_3^{(e)}(x) + a_4 \phi_4^{(e)}(x), \quad (30)$$

but now the  $a_i$  represent more than just the value of the approximate solution at each node, some also represent the value of its first derivative at each node,

$$\begin{aligned} \Psi^{(e)}(x_{1,3}) &= a_{1,3}, \\ \Psi^{(e)}(x_{2,4}) &= a_{2,4}. \end{aligned} \quad (31)$$

The shape functions satisfy an interpolation property of the form

$$\begin{aligned} \phi_1^{(e)}(x_1) &= 1, & \phi_2^{(e)}(x_1) &= 0, & \phi_3^{(e)}(x_1) &= 0, & \phi_4^{(e)}(x_1) &= 0, \\ \phi_1^{(e)}(x_1) &= 0, & \phi_2^{(e)}(x_1) &= 1, & \phi_3^{(e)}(x_1) &= 0, & \phi_4^{(e)}(x_1) &= 0, \\ \phi_1^{(e)}(x_2) &= 0, & \phi_2^{(e)}(x_2) &= 0, & \phi_3^{(e)}(x_2) &= 1, & \phi_4^{(e)}(x_2) &= 0, \\ \phi_1^{(e)}(x_2) &= 0, & \phi_2^{(e)}(x_2) &= 0, & \phi_3^{(e)}(x_2) &= 0, & \phi_4^{(e)}(x_2) &= 1, \end{aligned} \quad (32)$$

and just as with quadratic elements, this interpolation property can be used to solve for each of the shape functions,

$$\begin{aligned} \phi_1^{(e)}(x) &= 2r^3 - 3r^2 + 1, \\ \phi_2^{(e)}(x) &= (r^3 - 2r^2 + r)(x_2 - x_1), \\ \phi_3^{(e)}(x) &= 3r^2 - 2r^3, \\ \phi_4^{(e)}(x) &= (r^3 - r^2)(x_2 - x_1), \end{aligned} \quad (33)$$

where

$$r = \frac{(x - x_1)}{(x_2 - x_1)}. \quad (34)$$

Shape functions of this type (derived from nodes with two degrees of freedom) are from a family of shape functions called the Hermitian interpolation polynomials. By using the mapping for two nodes per element, Eq. (16), the corresponding isoparametric shape functions can be found,

$$\begin{aligned}
\varphi_1^{(e)}(\xi) &= \frac{1}{4}(\xi-1)^2(\xi+2), \\
\varphi_2^{(e)}(\xi) &= \frac{1}{4}(\xi-1)^2(\xi+1), \\
\varphi_3^{(e)}(\xi) &= -\frac{1}{4}(\xi+1)^2(\xi-2), \\
\varphi_4^{(e)}(\xi) &= \frac{1}{4}(\xi+1)^2(\xi-1),
\end{aligned} \tag{35}$$

and the transformation function can be worked out from the usual equation,

$$x = \chi^{(e)}(\xi) = \sum_{k=1}^4 x_k^{(e)} \varphi_k^{(e)}(\xi). \tag{36}$$

While Eqs. (35) properly interpolate the local domain, they must be changed slightly so that the element equations, Eqs. (21), can be used for all isoparametric elements no matter what their order. Since  $\varphi_2^{(e)}$  and  $\varphi_4^{(e)}$  represent the a slope (essentially a derivative) at each nodal point, a Jacobian term needs to be included with each term, so finally

$$\begin{aligned}
\varphi_1^{(e)}(\xi) &= \frac{1}{4}(\xi-1)^2(\xi+2), \\
\varphi_2^{(e)}(\xi) &= \frac{1}{4}(\xi-1)^2(\xi+1)J^{(e)}(\xi), \\
\varphi_3^{(e)}(\xi) &= -\frac{1}{4}(\xi+1)^2(\xi-2), \\
\varphi_4^{(e)}(\xi) &= \frac{1}{4}(\xi+1)^2(\xi-1)J^{(e)}(\xi),
\end{aligned} \tag{37}$$

and they can be used in Eqs. (21) so that the element equations for elements containing two nodes with two degrees of freedom each can be derived.

### 3. Numerical integration

A number of isoparametric elements have been derived, and the element equations for each type have reduced to Eqs. (21), which are reprinted here,

$$\begin{aligned}
K\alpha_{ij}^{(e)} &= \int_{-1}^{+1} \alpha(\chi^{(e)}(\xi)) \frac{1}{J^{(e)}(\xi)} \varphi_i'(\xi) \frac{1}{J^{(e)}(\xi)} \varphi_j'(\xi) J^{(e)}(\xi) d\xi, \\
K\beta_{ij}^{(e)} &= \int_{-1}^{+1} \beta(\chi^{(e)}(\xi)) \varphi_i(\xi) \varphi_j(\xi) J^{(e)}(\xi) d\xi, \\
M_{ij}^{(e)} &= \int_{-1}^{+1} \gamma(\chi^{(e)}(\xi)) \varphi_i(\xi) \varphi_j(\xi) J^{(e)}(\xi) d\xi.
\end{aligned} \tag{38}$$

One of the major advantages of isoparametric elements is that they arrive at element equations of this form. These equations lend themselves very well to numerical integration, which is needed if these elements are to be fully implemented into an FEM code. Applying Gaussian-Legendre quadrature, Eqs. (38) become

$$\begin{aligned} K\alpha_{ij}^{(e)} &\equiv \sum_{k=1}^n w_{nk} \alpha(\chi^{(e)}(\xi_{nk})) \frac{1}{J^{(e)}(\xi_{nk})} \phi_i'(\xi_{nk}) \frac{1}{J^{(e)}(\xi_{nk})} \phi_j'(\xi_{nk}) J^{(e)}(\xi_{nk}), \\ K\beta_{ij}^{(e)} &\equiv \sum_{k=1}^n w_{nk} \beta(\chi^{(e)}(\xi_{nk})) \phi_i(\xi_{nk}) \phi_j(\xi_{nk}) J^{(e)}(\xi_{nk}), \\ M_{ij}^{(e)} &\equiv \sum_{k=1}^n w_{nk} \gamma(\chi^{(e)}(\xi_{nk})) \phi_i(\xi_{nk}) \phi_j(\xi_{nk}) J^{(e)}(\xi_{nk}), \end{aligned} \quad (39)$$

where  $w_{nk}$  are the weighting factors, and  $\xi_{nk}$  are the corresponding Gaussian quadrature points.

For Gaussian-Legendre quadrature, an  $n$ -degree quadrature will exactly integrate a polynomial integrand of up to degree  $2n-1$ . Recall that  $\alpha$ ,  $\beta$ , and  $\gamma$  are quadratic polynomials in  $x$ ,  $\chi^{(e)}(\xi_{nk})$  and  $\phi_i(\xi_{nk})$  are polynomials in  $\xi$ , and  $J^{(e)}(\xi_{nk})$  is a constant, resulting in a polynomial of  $\xi$  to some degree. So for the current problem, a Gaussian-Legendre quadrature of high enough degree will perfectly integrate the element equation entries, Eqs. (39).

### C. ASSEMBLY

For each element in the domain a set of element equations can now be formulated. The matrix form for each set of element equations, Eqs. (10), is reprinted here

$$[K]^{(e)} \{a\} - \lambda [M]^{(e)} \{a\} = \{0\}, \quad (40a)$$

where

$$[K]^{(e)} = [K\alpha]^{(e)} + [K\beta]^{(e)}, \quad (40b)$$

with the entries determined numerically from Eqs. (39). After the element equations are determined for each element in the domain, these element equations need to be assembled into the final system equations,

$$[K]\{a\} - \lambda[M]\{a\} = \{0\}, \quad (41)$$

one large coupled system of equations that can be used to determine each of the eigenvalues,  $\lambda$ , and the eigenvectors,  $a_i$ .

Consider a domain containing three quadratic elements. Each element contains three nodes, with nodes on inter-element boundaries shared between two elements. The element equations for the three elements are

$$\begin{aligned} & \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix}^{(1)} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} - \lambda \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}^{(1)} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}, \\ & \begin{bmatrix} K_{33} & K_{34} & K_{35} \\ K_{43} & K_{44} & K_{45} \\ K_{53} & K_{54} & K_{55} \end{bmatrix}^{(2)} \begin{Bmatrix} a_3 \\ a_4 \\ a_5 \end{Bmatrix} - \lambda \begin{bmatrix} M_{33} & M_{34} & M_{35} \\ M_{43} & M_{44} & M_{45} \\ M_{53} & M_{54} & M_{55} \end{bmatrix}^{(2)} \begin{Bmatrix} a_3 \\ a_4 \\ a_5 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}, \\ & \begin{bmatrix} K_{55} & K_{56} & K_{57} \\ K_{65} & K_{66} & K_{67} \\ K_{75} & K_{76} & K_{77} \end{bmatrix}^{(3)} \begin{Bmatrix} a_5 \\ a_6 \\ a_7 \end{Bmatrix} - \lambda \begin{bmatrix} M_{55} & M_{56} & M_{57} \\ M_{65} & M_{66} & M_{67} \\ M_{75} & M_{76} & M_{77} \end{bmatrix}^{(3)} \begin{Bmatrix} a_5 \\ a_6 \\ a_7 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix}, \end{aligned} \quad (42)$$

which lead to the assembled system equations

$$\begin{aligned}
& \begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & & & & \\ K_{21}^{(1)} & K_{22}^{(1)} & K_{23}^{(1)} & & & & \\ K_{31}^{(1)} & K_{32}^{(1)} & K_{33}^{(1)} + K_{33}^{(2)} & K_{34}^{(2)} & K_{35}^{(2)} & & \\ & & K_{43}^{(2)} & K_{44}^{(2)} & K_{45}^{(2)} & & \\ & & K_{53}^{(2)} & K_{54}^{(2)} & K_{55}^{(2)} + K_{55}^{(3)} & K_{56}^{(3)} & K_{57}^{(3)} \\ & & & & K_{65}^{(3)} & K_{66}^{(3)} & K_{67}^{(3)} \\ & & & & K_{75}^{(3)} & K_{76}^{(3)} & K_{77}^{(3)} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{Bmatrix} \\
& - \lambda \begin{bmatrix} M_{11}^{(1)} & M_{12}^{(1)} & M_{13}^{(1)} & & & & \\ M_{21}^{(1)} & M_{22}^{(1)} & M_{23}^{(1)} & & & & \\ M_{31}^{(1)} & M_{32}^{(1)} & M_{33}^{(1)} + M_{33}^{(2)} & M_{34}^{(2)} & M_{35}^{(2)} & & \\ & & M_{43}^{(2)} & M_{44}^{(2)} & M_{45}^{(2)} & & \\ & & M_{53}^{(2)} & M_{54}^{(2)} & M_{55}^{(2)} + M_{55}^{(3)} & M_{56}^{(3)} & M_{57}^{(3)} \\ & & & & M_{65}^{(3)} & M_{66}^{(3)} & M_{67}^{(3)} \\ & & & & M_{75}^{(3)} & M_{76}^{(3)} & M_{77}^{(3)} \end{bmatrix} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}. \tag{43}
\end{aligned}$$

Now consider a domain containing two elements. Each element in this domain will contain two nodes with two degrees of freedom per node. The element equations for the two elements are

$$\begin{aligned}
& \begin{bmatrix} K_{11} & K_{12} & K_{13} & K_{14} \\ K_{21} & K_{22} & K_{23} & K_{24} \\ K_{31} & K_{32} & K_{33} & K_{34} \\ K_{41} & K_{42} & K_{43} & K_{44} \end{bmatrix}^{(1)} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{Bmatrix} - \lambda \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}^{(1)} \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}, \\
& \begin{bmatrix} K_{33} & K_{34} & K_{35} & K_{36} \\ K_{43} & K_{44} & K_{45} & K_{46} \\ K_{53} & K_{54} & K_{55} & K_{56} \\ K_{63} & K_{64} & K_{65} & K_{66} \end{bmatrix}^{(2)} \begin{Bmatrix} a_3 \\ a_4 \\ a_5 \\ a_6 \end{Bmatrix} - \lambda \begin{bmatrix} M_{33} & M_{34} & M_{35} & M_{36} \\ M_{43} & M_{44} & M_{45} & M_{46} \\ M_{53} & M_{54} & M_{55} & M_{56} \\ M_{63} & M_{64} & M_{65} & M_{66} \end{bmatrix}^{(2)} \begin{Bmatrix} a_3 \\ a_4 \\ a_5 \\ a_6 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}, \tag{44}
\end{aligned}$$

which lead to the assembled system equations

$$\begin{aligned}
& \left[ \begin{array}{cccccc}
K_{11}^{(1)} & K_{12}^{(1)} & K_{13}^{(1)} & K_{14}^{(1)} & & \\
K_{21}^{(1)} & K_{22}^{(1)} & K_{23}^{(1)} & K_{24}^{(1)} & & \\
K_{31}^{(1)} & K_{32}^{(1)} & K_{33}^{(1)} + K_{33}^{(2)} & K_{34}^{(1)} + K_{34}^{(2)} & K_{35}^{(2)} & K_{36}^{(2)} \\
& & K_{43}^{(1)} + K_{43}^{(2)} & K_{44}^{(1)} + K_{44}^{(2)} & K_{45}^{(2)} & K_{45}^{(2)} \\
& & K_{53}^{(2)} & K_{54}^{(2)} & K_{55}^{(2)} & K_{56}^{(2)} \\
& & K_{63}^{(2)} & K_{64}^{(2)} & K_{65}^{(2)} & K_{66}^{(2)}
\end{array} \right] \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{Bmatrix} \\
& -\lambda \left[ \begin{array}{cccccc}
M_{11}^{(1)} & M_{12}^{(1)} & M_{13}^{(1)} & M_{14}^{(1)} & & \\
M_{21}^{(1)} & M_{22}^{(1)} & M_{23}^{(1)} & M_{24}^{(1)} & & \\
M_{31}^{(1)} & M_{32}^{(1)} & M_{33}^{(1)} + M_{33}^{(2)} & M_{34}^{(1)} + M_{34}^{(2)} & M_{35}^{(2)} & M_{36}^{(2)} \\
& & M_{43}^{(1)} + M_{43}^{(2)} & M_{44}^{(1)} + M_{44}^{(2)} & M_{45}^{(2)} & M_{45}^{(2)} \\
& & M_{53}^{(2)} & M_{54}^{(2)} & M_{55}^{(2)} & M_{56}^{(2)} \\
& & M_{63}^{(2)} & M_{64}^{(2)} & M_{65}^{(2)} & M_{66}^{(2)}
\end{array} \right] \begin{Bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}. \tag{45}
\end{aligned}$$

Notice that for both sets of system equations there are  $a_i$  that are common at inter-element boundaries. This results in the overlapping within the K and M matrices. Only two examples have been illustrated, but all types of element equations are assembled in this fashion.

Finally these assembled system equations can be sent to a numerical routine that computes all of the eigenvalues and eigenvectors of a generalized eigenproblem. Such standard routines exist in the LAPACK<sup>5</sup> collection.



### III. THE ONE-DIMENSIONAL CODE

Now that the theory for the general, one-dimensional eigenproblem, Eq. (2), has been reviewed, the code that I developed to facilitate in the finite element solutions of such eigenproblems needs to be introduced. The complete listing of the eigenproblem code is called TISEFEA (Time-Independent Schrödinger Equation, Finite Element Analysis) and is located in the Appendix. Also located in the Appendix is the user-input data format.

I developed TISEFEA in object-oriented C++ using Microsoft Visual C++ 6.0. It uses various numerical routines from the proven LAPACK software library while it also utilizes the more recently developed Template Numerical Toolkit (TNT).<sup>6</sup> TNT is a collection of mathematical libraries for numerical computation in C++ whose fundamental classes include vectors, matrices, and multidimensional arrays. As Figure 5 shows, TISEFEA has a very simple object-oriented design, in fact, there are only three classes in the design, **Model**, **Element**, and **Shapes** (this object design does not include the various TNT vector and matrix classes used throughout the code, as they were not developed by the author and were utilized as standard data types). For reference, Table 1 also shows the public interface for each class.

Figure 5 – TISEFEA object model.

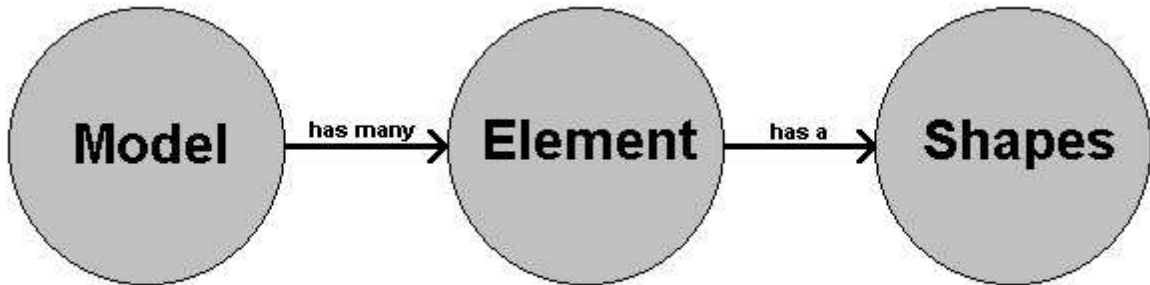


Table 1 – Public interfaces

Model		Element		Shapes
GetModelData Solve OutputSolution		SetElementData LeftBound RightBound Length NumberOfNodes DofPerNode AssembleElement K M		SetShapes X Phi DPhi Jacobian

**Shapes** objects are used to hold a collection of shape data for various types of isoparametric elements. This data includes the Jacobian, the shape functions, the shape function derivatives, and the coordinate transformation functions. Each **Element** object creates one instance of a **Shapes** object and initializes the **Shapes** object by calling the **SetShapes** routine. The **SetShapes** routine lets the **Shapes** object know what type and size of isoparametric element the **Element** object is so that the proper shape function and other information can be retrieved from the **Shapes** object.

An array of **Element** objects is contained in the **Model** object. The dimensions of each **Element** object are initialized by a call to **SetElementData**. The element

equations for each **Element** object can then be assembled with a call to **AssembleElement**, and finally the element equation matrices, **K** and **M**, for each element can be successfully accessed. The **Element** objects use twelve-point Gaussian quadrature to numerically integrate the entries for the element equations.

**Model** is obviously the highest class in the object design. One **Model** object is instantiated by the **main** routine, and this one **Model** object then acts as the interface between the **main** routine and the finite element analysis. The **Model** object has only three public interface routines, **GetModelData**, **Solve**, and **OutputSolution**, and the **main** routine calls each one in this order. **GetModelData** instructs the **Model** object to get the model data from the user and then initialize each of the elements. A call to the **Solve** routine instructs the **Model** object to assemble each of the element equations, assemble the completed system equations, and then solve the system equations with a call to a LAPACK routine. **OutputSolution** then outputs the desired solution data.

#### IV. BOUND SYSTEMS IN ONE DIMENSION

##### A. HARMONIC OSCILLATOR

As an example problem, consider the finite element analysis of the quantum mechanical, simple harmonic oscillator. Such a system must obey Schrödinger's equation for the harmonic oscillator,

$$-\frac{\hbar^2}{2m}\Psi''(x) + \frac{1}{2}kx^2\Psi(x) = E\Psi(x), \quad (46)$$

with the boundary conditions

$$\Psi(-\infty) = 0 \quad \text{and} \quad \Psi(+\infty) = 0. \quad (47)$$

Utilizing a convenient change of variables casts this equation into the dimensionless and more computationally friendly form,

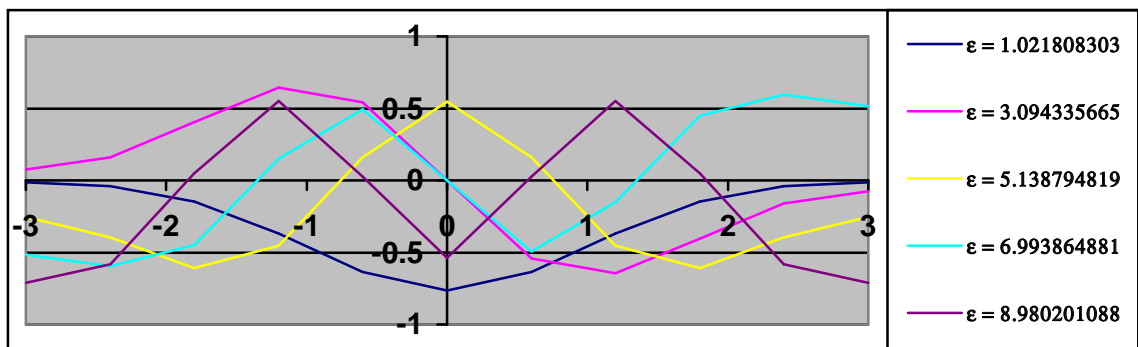
$$-\Psi''(\theta) + \theta^2 \Psi(\theta) = \varepsilon \Psi(\theta), \quad (48)$$

with similar boundary conditions as above. This equation fits the form of the general eigenproblem, Eq. (2), with  $\alpha = 1$ ,  $\beta = \theta^2$ , and  $\gamma = 1$ . Here the domain of such a problem would seem to be the entire  $\theta$  axis, however TISEFEA cannot handle an infinite domain, so the truncated domain,  $a < \theta < b$ , must be used. Sufficient values for  $a$  and  $b$  must be chosen so that  $\Psi(a)$  and  $\Psi(b)$  are negligibly small. And since good values for  $a$  and  $b$  are typically not known before numerical analysis, a trial and error approach must be undertaken.

Let's begin the investigation by considering a domain of  $-3 < \theta < +3$ , filled with 10 elements of type 2-1. An element of type 2-1 is a linear element with 1 degree of

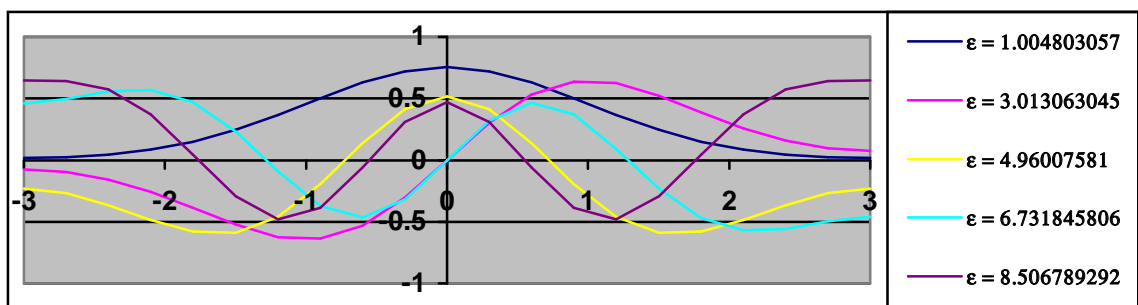
freedom at each node. (From now on elements will be referred to as of type p-q, where p is the number of nodes per element and q is the degree of freedom of each node.) Figure 6 depicts the solution to this model.

**Figure 6 – Harmonic oscillator eigenfunctions with 10 elements of type 2-1 with a domain from  $-3$  to  $3$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



Notice that the eigenfunctions do not look very smooth and that the eigenvalues are only close enough to the correct values to detect a proper trend. The correct eigenvalues are the odd integers 1, 3, 5, 7, 9, etc....

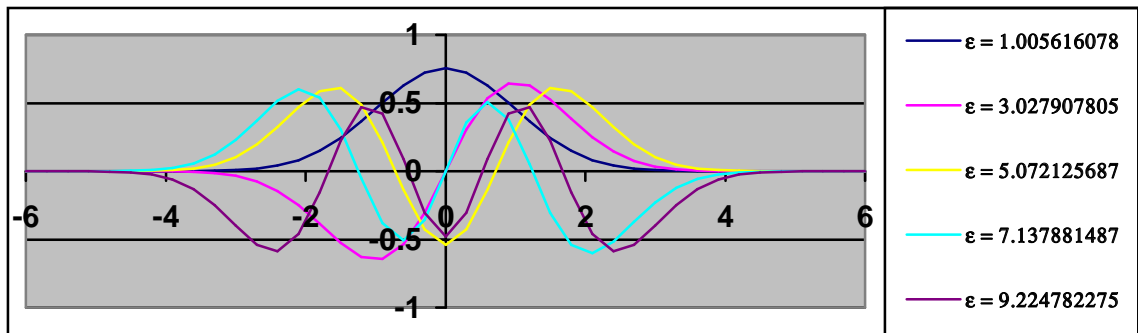
**Figure 7 - Harmonic oscillator eigenfunctions with 20 elements of type 2-1 with a domain from  $-3$  to  $3$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



Twice the number of elements is shown in Figure 7. In this case the eigenfunctions now seem smoother, but the eigenvalues do not really seem any closer to their correct values. Perhaps the domain is not large enough to give the correct values, so

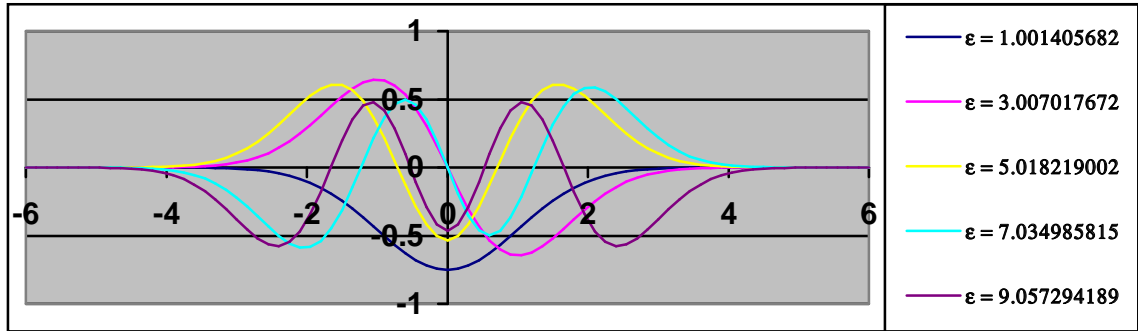
Figure 8 shows the solution for a domain of  $-6 < \theta < +6$ , filled with 40 elements of type 2-1.

**Figure 8 - Harmonic oscillator eigenfunctions with 40 elements of type 2-1 with a domain from  $-6$  to  $6$ . The legend shows the corresponding eigenvalue for each eigenfunction.**

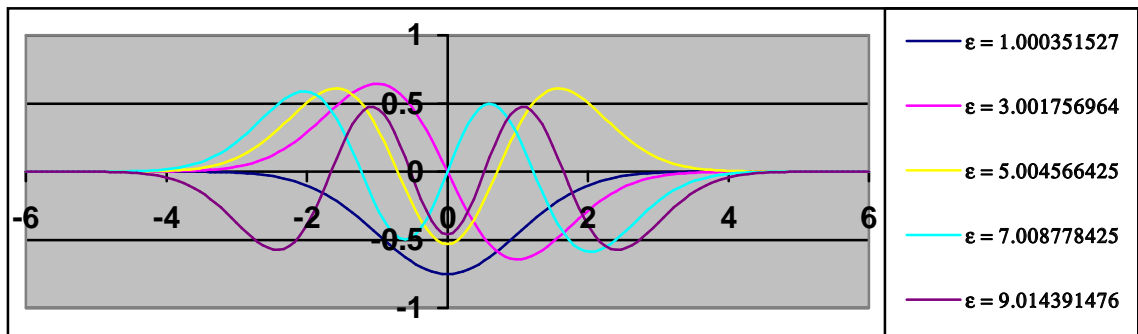


Notice that the new domain is clearly large enough, yet the eigenvalues do not seem to be converging all that fast. Let's try doubling the number of elements twice more.

**Figure 9 - Harmonic oscillator eigenfunctions with 80 elements of type 2-1 with a domain from  $-6$  to  $6$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



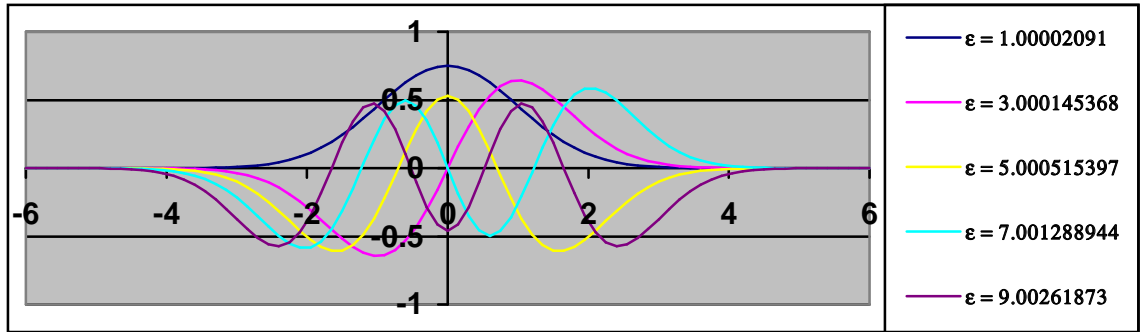
**Figure 10 - Harmonic oscillator eigenfunctions with 160 elements of type 2-1 with a domain from  $-6$  to  $6$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



As seen in Figure 9 and Figure 10, the eigenvalues finally seem to be getting closer to the correct values and the eigenfunctions are noticeably smoother than before.

Refining the model by increasing the number of elements within a domain, as we were doing here, is typically referred to as h-refinement. But h-refinement is only one of the systematic ways to get better results from a finite element model. Another technique is called p-refinement. In p-refinement, instead of increasing the number of elements in the domain, the number of nodes per each element is increased. For instance, if from Figure 8 the number of nodes per element had been increased to 3, the result would have been Figure 11. Look at the dramatic convergence of the eigenvalues.

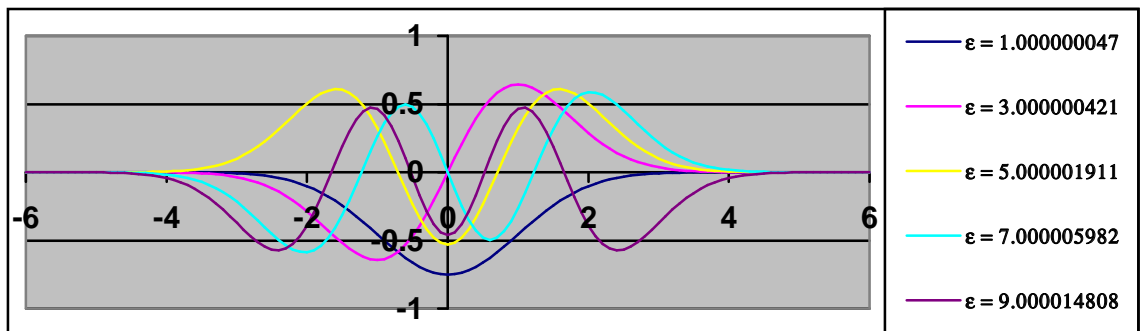
**Figure 11 - Harmonic oscillator eigenfunctions with 40 elements of type 3-1 with a domain from  $-6$  to  $6$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



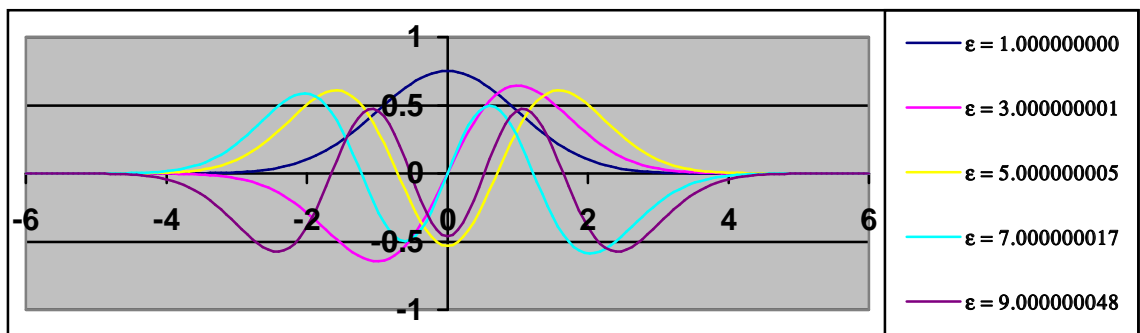
This convergence will seem even more dramatic after looking at Figure 12 and Figure 13.

In Figure 13 the first eigenvalue is accurate all the way out to nine decimal places.

**Figure 12 - Harmonic oscillator eigenfunctions with 40 elements of type 4-1 with a domain from  $-6$  to  $6$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



**Figure 13 - Harmonic oscillator eigenfunctions with 40 elements of type 5-1 with a domain from  $-6$  to  $6$ . The legend shows the corresponding eigenvalue for each eigenfunction.**



There are also elements with higher order nodes, such as elements of type 2-2.

Table 2 lists some of the first ten eigenvalues found by all the various elements



programmed into TISEFEA. As the shapes for the first five are clearly known from previous figures, the eigenfunctions are not shown. The domain used in Table 2 is  $-9 < \theta < +9$ . Notice that the accuracy increases with the number of nodes per element and the degree of freedom per node. This trend will typically continue until numerical error begins to exceed the benefits of higher order elements. For the harmonic oscillator, 90 elements of type 2-3 in the domain  $-9 < \theta < +9$  have successfully produced the first ten eigenvalues to nine decimal places.

**Table 2 - Harmonic oscillator eigenvalues for some elements of various type with domain from  $-9$  to  $9$ .**

Eigenvalue Number	Element Type					
	2-1	3-1	4-1	5-1	2-2	2-3
	30 Elements					
1	1.022367970	1.000326044	1.000002924	1.000000019	1.000006744	1.000000000
2	3.109021719	3.002221529	3.000025880	3.000000204	3.000055620	3.000000005
3	5.276232977	5.007709018	5.000115756	5.000001116	5.000230428	5.000000037
8	17.06219999	15.15080796	15.00572413	15.00011970	15.00801155	15.00001265
9	19.54560114	17.21269922	17.00883731	17.00022649	17.01184252	17.00002370
10	22.04429846	19.27648744	19.01445423	19.00032939	19.01812283	19.00005975
	90 Elements					
1	1.001405682	1.000004150	1.000000004	1.000000000	1.000000013	1.000000000
2	3.007017672	3.000028964	3.000000037	3.000000000	3.000000116	3.000000000
3	5.018219002	5.000103108	5.000000169	5.000000000	5.000000521	5.000000000
8	15.15722849	15.00233017	15.00000916	15.00000002	15.00002583	15.00000000
9	17.20146257	17.00336361	17.00001487	17.00000004	17.00004122	17.00000000
10	19.25111761	19.00466317	19.00002293	19.00000007	19.00006246	19.00000000

Having completed the example investigation into the finite element analysis of the quantum mechanical, simple harmonic oscillator, results are now found for a particle in a finite well, a particle in an infinite well, and the radial eigensolutions to the hydrogen atom.

## B. PARTICLE IN A FINITE WELL

Such a system obeys the time-independent, Schrödinger equation,

$$-\frac{\hbar^2}{2m}\Psi''(x) + V(x)\Psi(x) = E\Psi(x), \quad (49)$$

where

$$V(x) = \begin{cases} V_0 & -a/2 < x < a/2 \\ 0 & |x| > a/2 \end{cases}. \quad (50)$$

Only a finite number of the energy eigenvalues,  $E$ , will be bound. Employing a change of scale similar to the harmonic oscillator,

$$-\Psi''(\theta) + V(\theta)\Psi(\theta) = \varepsilon\Psi(\theta), \quad (51)$$

with

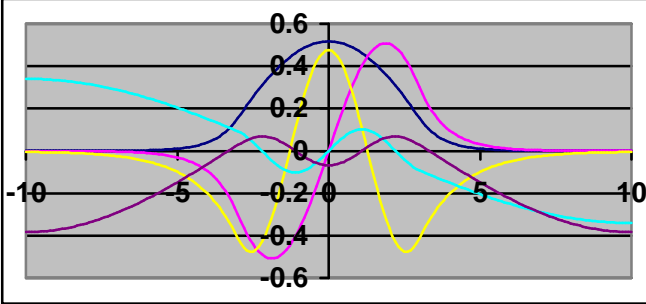
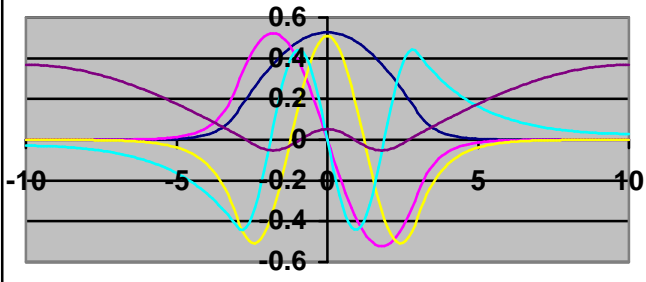
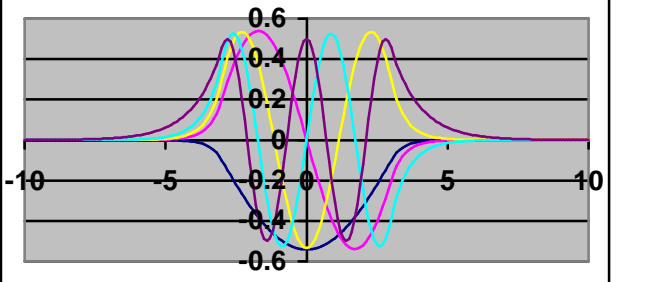
$$V(\theta) = \begin{cases} 0 & -a/2 < \theta < a/2 \\ V_0 & |\theta| > a/2 \end{cases}. \quad (52)$$

This equation fits the form of the general eigenproblem, Eq. (2), with  $\alpha = 1$ ,  $\beta = V(\theta)$ , and  $\gamma = 1$ . There are clearly three distinct regions within the infinite domain. Again this infinite domain will have to be approximated within TISEFEA by an appropriate finite domain and only a finite number of the energy eigenvalues,  $\varepsilon$ , will be bound.

Table 3 shows the TISEFEA results for 100 elements of type 2-3 over the domain  $-10 < \theta < +10$  with barrier width 6 and varying barrier heights. Notice that for a barrier height of 2 only three of the eigensolutions are bound. TISEFEA makes an attempt to solve for all the eigensolutions, but the boundary conditions coded into TISEFEA are only sufficient for bound eigensolutions, so any eigensolution that is not bound is an invalid result. When the barrier height is increased to 3 scaled units the fourth

eigensolution gets closer to being bound. With a barrier height of 6, the fourth and fifth eigensolutions are bound and valid.

**Table 3 – Particle in a finite well of barrier width 6 with domain from –10 to 10. There are 100 elements of type 2-3.**

Barrier height 2		
	0.178486359	
	0.69937968	
	1.494960485	
	2.034338177	
	2.05503352	
		Eigenvalues
	1	0.1784863588
	2	0.6993796801
Barrier height 3		
	0.192122361	
	0.759437711	
	1.665558866	
	2.769549088	
	3.045922002	
		Eigenvalues
	1	0.1921223614
	2	0.7594377114
Barrier height 6		
	0.212107599	
	0.844565016	
	1.883854421	
	3.297930906	
	4.995001748	
		Eigenvalues
	1	0.2121075994
	2	0.844565016

### C. PARTICLE IN AN INFINITE WELL

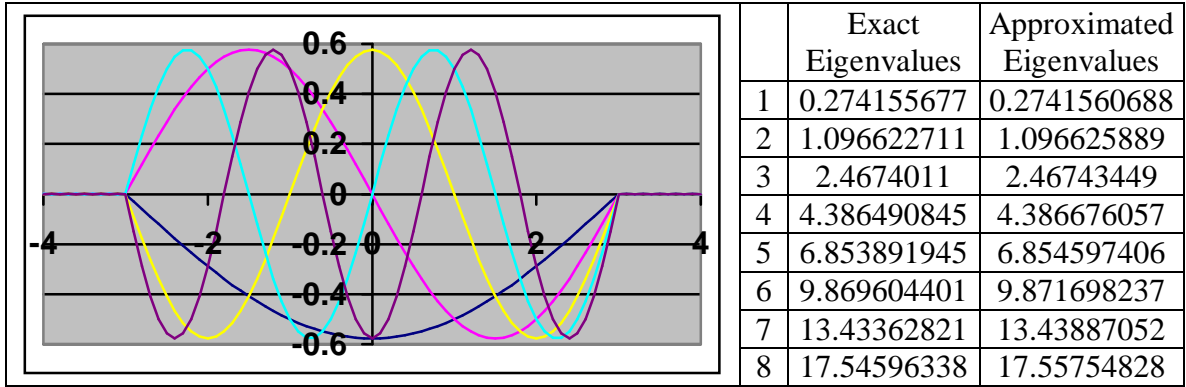
A particle in an infinite well can be treated with the same approach as a particle in a finite well. For a good approximation of an infinite well, however, the potential,  $V_0$ ,

must be taken very large. The exact eigenvalues for the dimensionless time-independent, Schrödinger equation, Eq. (51), with barriers of infinite height are

$$\varepsilon = \frac{\pi^2 n^2}{a^2}, \quad (53)$$

where  $n$  is an integer greater than, or equal to, one. Table 4 shows a TISEFEA solution to the particle in an infinite well with infinity approximated by the large number, 1E10.

**Table 4 – Particle in an approximated infinite well of barrier height 1E10, barrier width 6, and domain from -4 to 4. There are 40 elements of type 3-1.**



#### D. HYDROGEN ATOM ENERGY LEVELS

The time-independent, Schrödinger equation for an electron moving in the Coulomb potential of the hydrogen atom is,

$$-\frac{\hbar^2}{2m} \Psi''(r) - \frac{e^2}{r} \Psi(r) = E \Psi(r), \quad (54)$$

where  $r$  ranges from 0 to infinity. Rescaling  $r$  leads to,

$$-\Psi''(\theta) - \frac{2}{\theta} \Psi(\theta) = \varepsilon \Psi(\theta), \quad (55)$$

but in this equation,  $\beta$ , is not a quadratic polynomial and can not be input into TISEFEA.

To remedy this, Eq. (55), is multiplied by  $\theta^2$ , resulting in

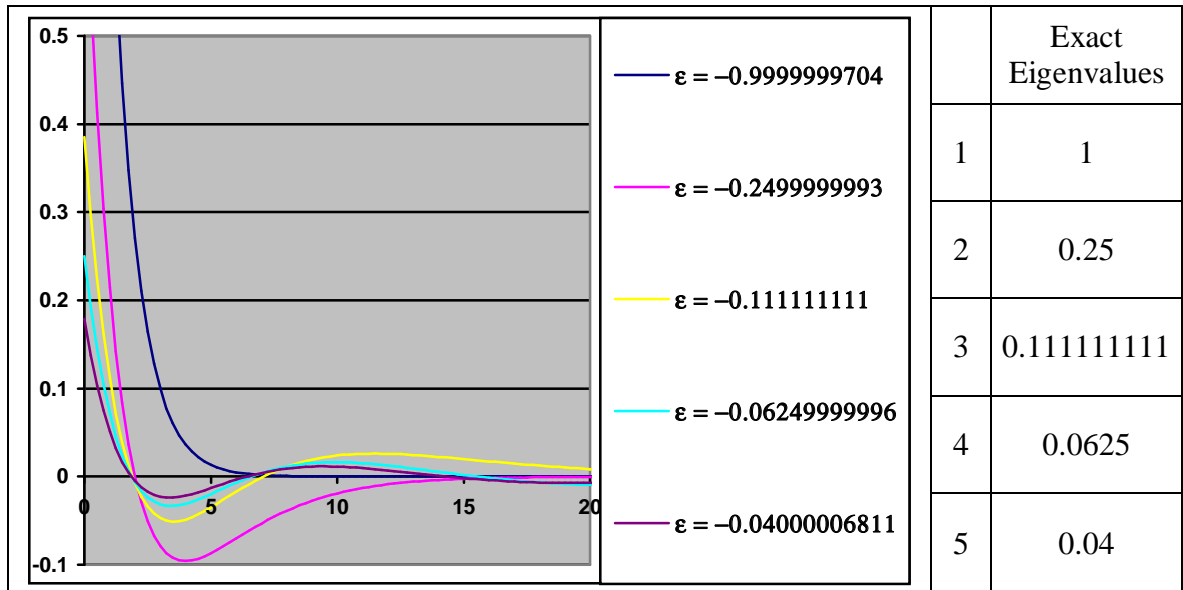
$$-\theta^2 \Psi''(\theta) - 2\theta \Psi'(\theta) = \varepsilon \theta^2 \Psi(\theta). \quad (56)$$

This equation now fits the form of the general one-dimensional eigenproblem, Eq. (2), with  $\alpha = \theta^2$ ,  $\beta = -2\theta$ , and  $\gamma = \theta^2$ . The exact eigenvalues are governed by the familiar

$$\varepsilon = -\frac{1}{n^2}, \quad (57)$$

where  $n$  is an integer greater than, or equal to, one. Table 5 shows a TISEFEA solution to the hydrogen atom. Again, the finite element analysis arrived at very precise results.

**Table 5 – Electron in the coulomb potential of the hydrogen atom. The solution domain is from 0 to 100 and there are 100 elements of type 5-1.**



## V. UNBOUND SYSTEMS IN ONE DIMENSION

An example of a quantum mechanical system that is not an eigenproblem, is quantum mechanical tunneling through a barrier. These systems obey the familiar time-independent, Schrödinger equation,

$$-\frac{\hbar^2}{2m}\Psi''(x) + V(x)\Psi(x) = E\Psi(x), \quad (58)$$

where  $E$  is now known. The change of scale results again in,

$$-\Psi''(\theta) + V(\theta)\Psi(\theta) = \varepsilon\Psi(\theta), \quad (59)$$

where  $V(\theta)$  and  $\varepsilon$  are both known. For quantum mechanical tunneling,  $V(\theta)$  typically has the form

$$V(\theta) = \begin{cases} V_0 & a < \theta < b \\ 0 & \theta < a, \quad \theta > b \end{cases}, \quad (60)$$

basically the opposite of the finite square well. The domain of the finite element analysis for such a problem will be restricted to  $a < \theta < b$ . This domain will then have the mixed boundary conditions

$$\begin{aligned} \Psi'(a) + ik_1\Psi(a) &= 2ik_1e^{ik_1a}, \\ \Psi'(b) - ik_1\Psi(b) &= 0. \end{aligned} \quad (61)$$

where

$$k_1 = \sqrt{\varepsilon}, \quad (62)$$

and

$$i = \sqrt{-1}. \quad (63)$$

These mixed boundary conditions are arrived at after realizing

$$\begin{aligned}
\Psi(a) &= e^{ik_1 a} + r e^{-ik_1 a}, \\
\Psi'(a) + ik_1 \Psi(a) &= ik_1 (e^{ik_1 a} - r e^{-ik_1 a}), \\
\Psi(b) &= t e^{ik_1 b}, \\
\Psi'(b) &= ik_1 t e^{ik_1 b},
\end{aligned} \tag{64}$$

where  $r$  and  $t$  are complex constants. If a set of elements is used that contains the first derivatives in the solution, such as elements of type 2-2, then these mixed boundary conditions can be resolved within the analysis and the Eqs. (64) can be arranged so that the transmission and reflection coefficients,

$$\begin{aligned}
R &= r^* r, \\
T &= t^* t,
\end{aligned} \tag{65}$$

can be found. For a description on how to apply these boundary conditions see the paper by Goloskie et. al.<sup>4</sup>

#### A. CHANGES TO THE ONE-DIMENSIONAL CODE

I made four major changes to TISEFEA. First, I changed the code to handle system equation entries that were of complex data type. Second, I added the assignment of the mixed boundary conditions. Third, the system equations were sent to a linear equation solver as opposed to an eigenproblem solver. And fourth, I resolved the transmission and reflection coefficients from the solution. These changes were made to the TISEFEA code with relative ease. (The specific changes to the code are not presented here, just the results.)

#### B. TUNNELING THROUGH A BARRIER

The exact solution for the transmission coefficient of a square barrier is

$$\frac{1}{T} = 1 + \frac{1}{4} \frac{V_0^2}{E(E - V_0)} \begin{cases} \sinh^2[k_2(b - a)], & V_0 > E \\ \sin^2[k_3(b - a)], & V_0 < E \end{cases}, \quad (66)$$

where

$$\begin{aligned} k_2 &= \sqrt{(V_0 - \varepsilon)}, \\ k_3 &= \sqrt{(\varepsilon - V_0)}, \end{aligned} \quad (67)$$

and the reflection coefficient is

$$R = 1 - T. \quad (68)$$

Table 6 shows a number of domains with varying particle energies, where the results are typically accurate to at least 5 decimal places.

**Table 6 – Tunneling through a square barrier of height 4. All elements are of type 2-2.**

Domain		Number of Elements	Particle Energy	Computed Transmission	Computed Reflection	Exact Transmission	Exact Reflection
a	b						
0	2	20	2	0.0138769	0.986093	0.01387683	0.986123169
0	2	40	2	0.0138768	0.986119	0.01387683	0.986123169
0	2	40	3	0.0539409	0.946058	0.05394086	0.94605914
0	2	40	3.5	0.104615	0.895384	0.104615427	0.895384573
0	1	20	3.5	0.426167	0.573832	0.42616745	0.57383255
0	0.5	10	3.5	0.770523	0.229477	0.770523327	0.229476673
0	2	40	3.95	0.187688	0.8123120	0.187688391	0.812311609
0	2	40	4	0.200000	0.8000000	0.200000000	0.800000000
0	2	40	4.25	0.272800	0.727200	0.272800073	0.727200073
0	2	40	5	0.601881	0.398119	0.601881198	0.398119198



## VI. A SIMPLE TWO-DIMENSIONAL FINITE ELEMENT METHOD

Here I present a technique for applying the finite element method in two dimensions using linear triangular elements. I integrated the technique into a new C++ computer code called, FEM2DLinear, and use this code to approximate eigensolutions to the isotropic two-dimensional harmonic oscillator, and the propagation of sound in a rectangular cavity. The general problem, then, is the numerical solution of an eigenproblem of the following type,

$$-\frac{\partial}{\partial x}\left(\alpha_x(x,y)\frac{\partial \Psi(x,y)}{\partial x}\right)-\frac{\partial}{\partial y}\left(\alpha_y(x,y)\frac{\partial \Psi(x,y)}{\partial y}\right) + \beta(x,y)\Psi(x,y) - \lambda\gamma(x,y)\Psi(x,y) = 0, \quad (69)$$

where  $\alpha_x$ ,  $\alpha_y$ ,  $\beta$ , and  $\gamma$  represent the physical properties and are all quadratic functions of  $x$  and  $y$  complete to degree 2, and  $\lambda$  is an unknown scalar. The eigenvalues,  $\lambda$ , and the eigenvectors,  $\Psi(x)$ , are sought. The domain under consideration here is any interval ( $a < x < b$ ,  $a' < y < b'$ ) with the following conditions on the boundary,

$$\Psi = 0 \quad \text{or} \quad \Psi' = 0. \quad (70)$$

As we proceed, one will note the many similarities between the two-dimensional and the one-dimensional code. This is consistent with the design of the FEM; once one understands the simplest techniques, it typically can be extended to more difficult techniques and systems.

### A. THE TWO-DIMENSIONAL TECHNIQUE

Paralleling the one-dimensional case, the domain of the problem must first be broken up into elements, the only main difference being that the domain is now over two dimensions. The approximate solution for the  $e^{\text{th}}$  element is then,

$$\Psi^{(e)}(x, y) = \sum_{j=1}^m a_j \phi_j^{(e)}(x, y), \quad (71)$$

where the shape functions are now two-dimensional. The approximate solution over the entire domain is then

$$\Psi(x, y) = \sum_{e=1}^n \Psi^{(e)}(x, y) = \sum_{e=1}^n \sum_{j=1}^m a_j \phi_j^{(e)}(x, y). \quad (72)$$

Utilizing the Galerkin method of weighted residuals, the following system for the  $e^{\text{th}}$  element is thus arrived at,

$$\begin{aligned} \int \int^{(e)} \left[ -\frac{\partial}{\partial x} \left( \alpha_x(x, y) \frac{\partial \Psi^{(e)}(x, y)}{\partial x} \right) - \frac{\partial}{\partial y} \left( \alpha_y(x, y) \frac{\partial \Psi^{(e)}(x, y)}{\partial y} \right) \right. \\ \left. + \beta(x, y) \Psi^{(e)}(x, y) - \lambda \gamma(x, y) \Psi^{(e)}(x, y) \right] \phi_i^{(e)}(x, y) dx dy = 0, \end{aligned} \quad (73)$$

where again  $i$  ranges from 1 to  $m$ . Integrating this to remove the second order terms, dropping the boundary terms that arrive, and substituting Eq. (71) in as the approximate solution for the  $e^{\text{th}}$  element, results in the element equations of the form

$$\sum_{j=1}^m \left[ \int \int^{(e)} \left[ \frac{\partial \phi_i^{(e)}}{\partial x} \alpha_x \frac{\partial \phi_j^{(e)}}{\partial x} \right] dx dy + \int \int^{(e)} \left[ \frac{\partial \phi_i^{(e)}}{\partial y} \alpha_y \frac{\partial \phi_j^{(e)}}{\partial y} \right] dx dy \right. \\ \left. + \int \int^{(e)} [\phi_i^{(e)} \beta \phi_j^{(e)}] dx dy - \lambda \int \int^{(e)} [\phi_i^{(e)} \gamma \phi_j^{(e)}] dx dy \right] a_j = 0. \quad (74)$$

Finally, the element equations can be written in the now familiar matrix form

$$[K]^{(e)} \{a\} - \lambda [M]^{(e)} \{a\} = \{0\}, \quad (75a)$$

where

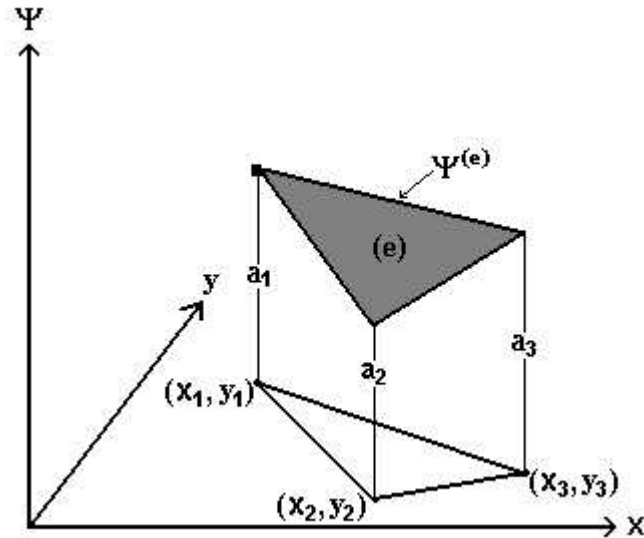
$$[K]^{(e)} = [K\alpha]^{(e)} + [K\beta]^{(e)}, \quad (75b)$$

and

$$\begin{aligned} K\alpha_{ij}^{(e)} &= \iint^{(e)} \left[ \frac{\partial \phi_i^{(e)}}{\partial x} \alpha_x \frac{\partial \phi_j^{(e)}}{\partial x} \right] dx dy + \iint^{(e)} \left[ \frac{\partial \phi_i^{(e)}}{\partial y} \alpha_y \frac{\partial \phi_j^{(e)}}{\partial y} \right] dx dy, \\ K\beta_{ij}^{(e)} &= \iint^{(e)} [\phi_i^{(e)} \beta \phi_j^{(e)}] dx dy, \\ M_{ij}^{(e)} &= \iint^{(e)} [\phi_i^{(e)} \gamma \phi_j^{(e)}] dx dy. \end{aligned} \quad (76)$$

To most easily understand this two-dimensional application of the FEM, I have chosen to use fairly simple linear triangular elements. The element trial solution for the  $e^{\text{th}}$  linear triangular element is depicted graphically in Figure 14, below. There are three nodes in each element and the trial solution for the  $e^{\text{th}}$  element is,

**Figure 14 – A linear triangular element**



$$\Psi^{(e)}(x, y) = \sum_{j=1}^3 a_j \phi_j^{(e)}(x, y), \quad (77)$$

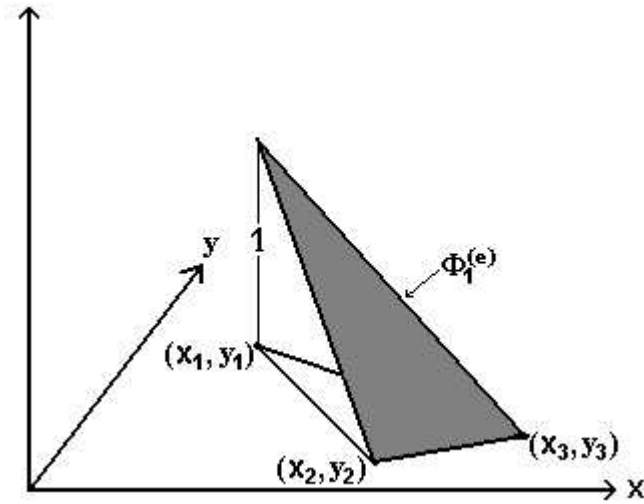
where the trial solution linearly interpolates the corner nodes and thus takes on the value  $a_i$  there,

$$\Psi^{(e)}(x_i, y_i) = a_i, \quad (78)$$

and the shape functions must again adhere to the interpolation property

$$\varphi_j^{(e)}(x_i, y_i) = \delta_{ij}. \quad (79)$$

**Figure 15 – Shape function for a linear triangular element**



Pictured in Figure 15 is the first shape function for the  $e^{\text{th}}$  element. The shape functions obviously take on the form of two-dimensional linear polynomials,

$$\varphi_j^{(e)}(x_i, y_i) = a_j + b_j x_i + c_j y_i. \quad (80)$$

Applying the interpolation property at each of the nodes, the exact form of the shape functions can be found,

$$\varphi_j^{(e)}(x_i, y_i) = \frac{a_j + b_j x_i + c_j y_i}{2 A^{(e)}}, \quad (81)$$

where

$$\begin{aligned}
a_j &= x_k y_l - x_l y_k, \\
b_j &= y_k - y_l, \\
c_j &= x_l - x_k,
\end{aligned} \tag{82}$$

and  $A^{(e)}$  is the area of the element (e),

$$A^{(e)} = \frac{1}{2} [(x_2 y_3 - x_3 y_2) - (x_1 y_3 - x_3 y_1) + (x_1 y_2 - x_2 y_1)]. \tag{83}$$

As a final step before resolving the element system equation matrix entries, Eqs. (76), the physical properties are evaluated at the centroid of each element. So for element (e),

$$\begin{aligned}
\alpha_x^{(e)} &= \alpha_x(x_c^{(e)}, y_c^{(e)}), \\
\alpha_y^{(e)} &= \alpha_y(x_c^{(e)}, y_c^{(e)}), \\
\beta^{(e)} &= \beta(x_c^{(e)}, y_c^{(e)}), \\
\gamma^{(e)} &= \gamma(x_c^{(e)}, y_c^{(e)}),
\end{aligned} \tag{84}$$

where

$$\begin{aligned}
x_c^{(e)} &= (x_1 + x_2 + x_3), \\
y_c^{(e)} &= (y_1 + y_2 + y_3).
\end{aligned} \tag{85}$$

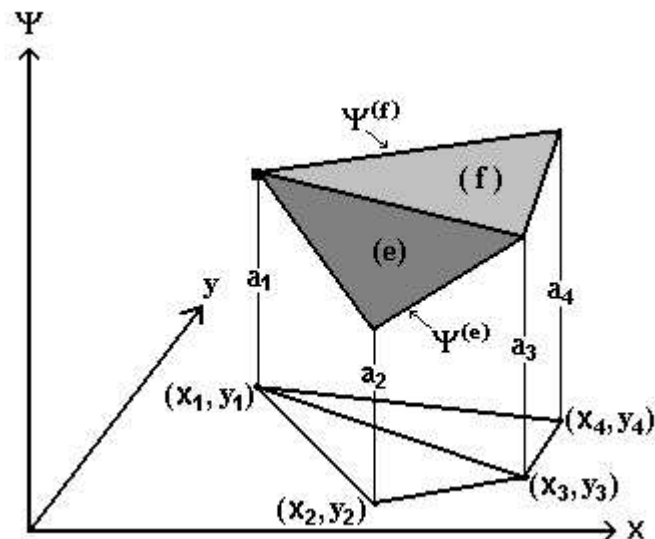
This approximation will obviously introduce some modeling error into the solution, however it can be made negligibly small by taking elements that are sufficiently small.

Now the integrals in the system matrix entries are resolved, resulting in

$$\begin{aligned}
K\alpha_{ij}^{(e)} &= \frac{\alpha_x^{(e)} b_i b_j + \alpha_y^{(e)} c_i c_j}{4A^{(e)}}, \\
K\beta_{ij}^{(e)} &= \begin{cases} \frac{\beta^{(e)} A^{(e)}}{6}, & i = j \\ \frac{\beta^{(e)} A^{(e)}}{12}, & i \neq j \end{cases}, \\
M_{ij}^{(e)} &= \begin{cases} \frac{\gamma^{(e)} A^{(e)}}{6}, & i = j \\ \frac{\gamma^{(e)} A^{(e)}}{12}, & i \neq j \end{cases}.
\end{aligned} \tag{86}$$

Since the integrals for the linear elements were fairly simple, and since I am only using this one type of element in my two-dimensional code, as opposed to the myriad of elements in the one-dimensional code, the use of numerical integrations would have been superfluous. Numerical integrations are only typically needed when a code is made general and applied to many different problems.

**Figure 16 – Two neighboring linear triangular elements**



Now that the form of the element equations has been found they need to be assembled into the final system equations. The technique of assembling the element equations for two dimensions is very similar to assembling the element equations in one dimension. Referring to Figure 16, nodes 1 and 3 will be part of the element equations for both elements (e) and (f), and so will overlap when assembling the final system equations, just as seen previously for one-dimensional elements. The resulting final system equations can be solved with the same general eigensolver from the LAPACK collection as was used in the one-dimensional case.

## B. TWO-DIMENSIONAL APPLICATIONS

### 1. Isotropic harmonic oscillator

The dimensionless equation for the two-dimensional isotropic harmonic oscillator is

$$-\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\Psi(x, y) + (x^2 + y^2)\Psi(x, y) = E \Psi(x, y), \quad (87)$$

where the integer eigenvalues are

$$E_n = n + 1, \quad (88)$$

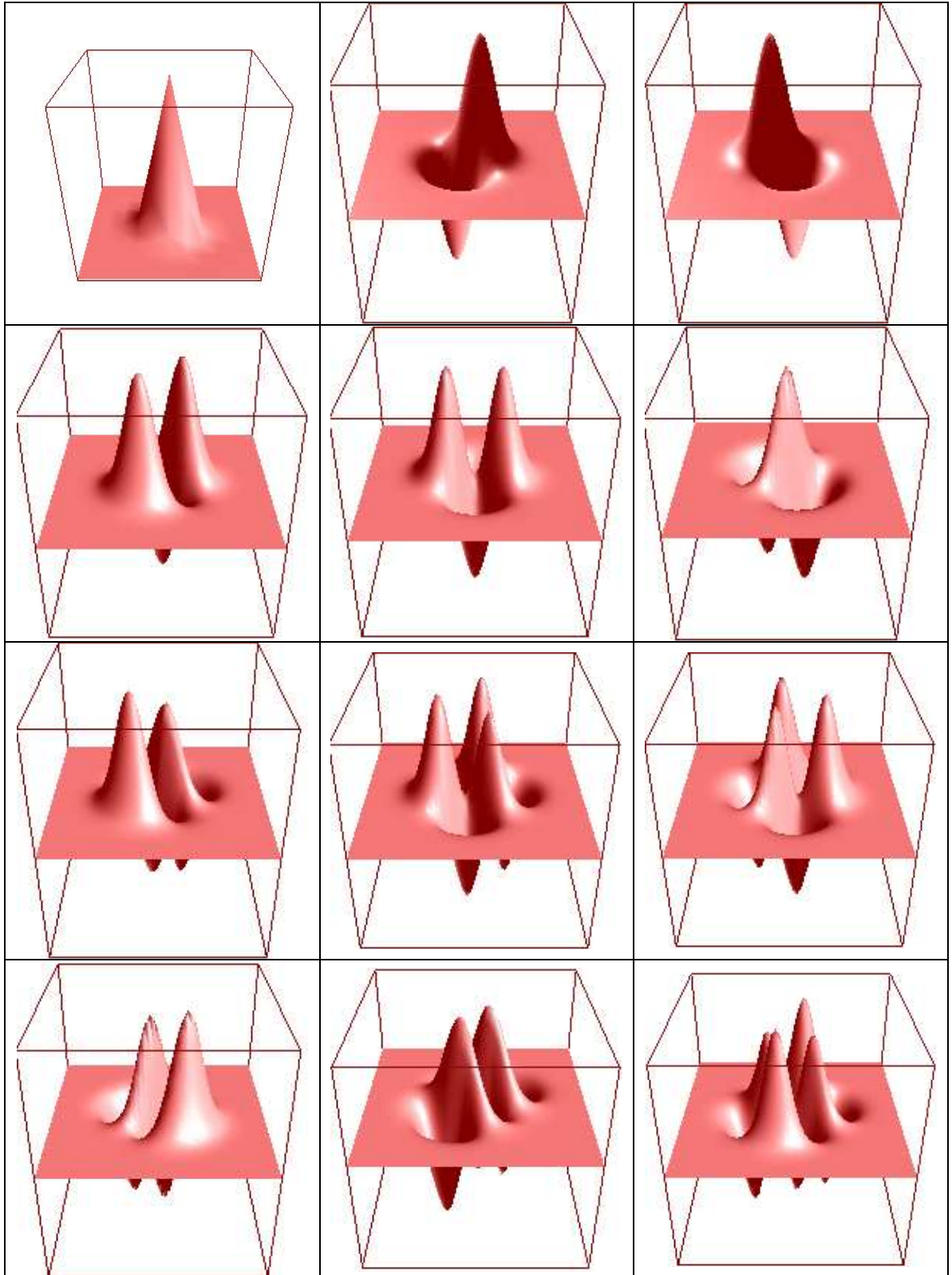
and the degeneracy of the  $n^{\text{th}}$  level is  $n + 1$ . Eq. (87) fits into the form of the general eigenproblem, Eq. (69), with

$$\begin{aligned} \alpha_x &= 1, \\ \alpha_y &= 1, \\ \beta &= x^2 + y^2, \\ \gamma &= 1. \end{aligned} \quad (89)$$

**Table 7 – Estimated eigenvalues,  $E_n$ , for the two-dimensional isotropic harmonic oscillator using linear triangular elements with  $K$  linear divisions over the domain  $(-6 < x < 6, -6 < y < 6)$ .**

	K = 10	K = 20	K = 30	K = 40	K = 50
n = 0	2.4757	2.11575	2.05124	2.02879	2.01841
n = 1	4.71264	4.17408	4.07755	4.04367	4.02797
	5.29568	4.34157	4.1548	4.08776	4.05639
n = 2	7.00243	6.27454	6.12321	6.06955	6.04459
	7.35801	6.42184	6.19277	6.1096	6.07051
	8.43614	6.75432	6.34999	6.20024	6.12923
n = 3	9.51327	8.41548	8.18785	8.10632	8.06824
	10.0852	8.54347	8.24987	8.14234	8.09165
	10.985	8.85003	8.39847	8.22876	8.14786
	11.7532	9.33592	8.63216	8.36453	8.23617
n = 4	11.8195	10.5952	10.2711	10.1538	10.0988
	12.0388	10.7049	10.3258	10.1859	10.1198
	12.1982	10.9865	10.4661	10.2682	10.1735
	13.6656	11.4396	10.6896	10.3992	10.2591
	14.5248	12.0714	10.9974	10.5792	10.3766

**Table 8 - Estimated eigenfunctions for the two-dimensional isotropic harmonic oscillator using linear triangular elements with 50 linear divisions over the domain  $(-6 < x < 6, -6 < y < 6)$ .**





Using the results from the one-dimensional case to help here, we know that a sufficient domain for this problem is  $(-6 < x < 6, -6 < y < 6)$ . The domain is broken up into right isosceles triangles by first dividing the domain into a square grid with  $K$  linear divisions and then dividing each square along a diagonal. Table 7 shows the approximated eigenvalues through  $n = 4$  for varying number of linear divisions, while Table 8 shows the first few corresponding (unnormalized) eigenfunctions,  $\Psi_n$ . I incorporated some OpenGL® API routines into the code to draw the plots.

## 2. Propagation of sound in a rectangular cavity

Using separation of variables on the two-dimensional wave equation leads to the two-dimensional Helmholtz equation,

$$-\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\Psi(x, y) + \mu^2 \Psi(x, y) = 0, \quad (90)$$

which can provide, among other things, some information about the propagation of sound in a rectangular cavity. In particular, it can provide the modes of propagation,  $\Psi_n$ , and the corresponding cut-off frequencies for these modes,

$$f_n = \frac{\mu_n c}{2\pi}, \quad (91)$$

where  $c = 1127$  ft/sec, the approximate speed of sound in air, and

$$-\mathbf{n} \cdot \left[ \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \Psi(x, y) \right] = 0 \quad (92)$$

on the boundary of the domain ( $\mathbf{n}$  is the unit normal to the domain boundary). Eq. (90)

fits into the form of the general eigenproblem, Eq. (69), with

$$\begin{aligned}
\alpha_x &= 1, \\
\alpha_y &= 1, \\
\beta &= 0, \\
\gamma &= 1.
\end{aligned} \tag{93}$$

Here I consider a domain of  $(-12 < x < 12, -14 < y < 14)$  with K linear divisions. Then the exact solution to the cut-off frequencies is,

$$f^{(pq)} = \frac{c}{2} \sqrt{\left(\frac{p}{24}\right)^2 + \left(\frac{q}{28}\right)^2} \quad p, q = 0, 1, 2, \dots, \tag{94}$$

and the exact solution for the eigenfunctions is

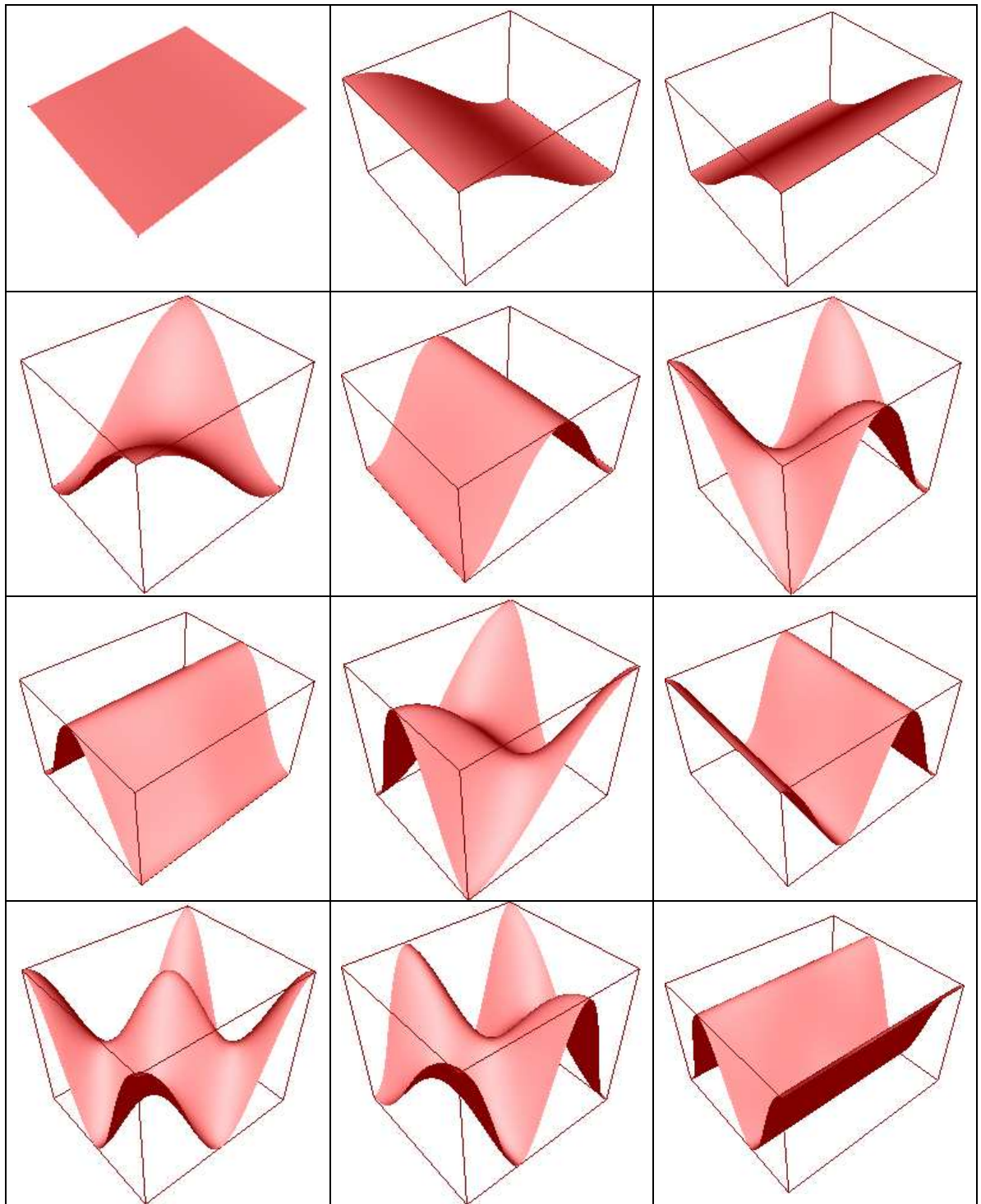
$$\Psi^{(pq)}(x, y) = P \cos\left(\frac{\pi p x}{24}\right) \cos\left(\frac{\pi q y}{28}\right) \quad p, q = 0, 1, 2, \dots, \tag{95}$$

where P is an arbitrary amplitude. Table 9 shows the first 12 approximated cut-off frequencies,  $f_n$ , for varying number of linear divisions, while Table 8 shows the first 12 (unnormalized) eigenfunctions,  $\Psi_n$ .

**Table 9 - Estimated cut-off frequencies,  $f_n$ , for the propagation of sound in a rectangular cavity using linear triangular elements with K linear divisions over  $(-12 < x < 12, -14 < y < 14)$ .**

n	Exact	K = 10	K = 20	K = 30	K = 40	K = 50
1	0.0000	3.79E-06	8.43E-06	4.56E-06	7.71E-06	1.67E-05
2	20.1250	20.2069	20.1456	20.1342	20.1302	20.1283
3	23.4792	23.5744	23.5032	23.4899	23.4852	23.483
4	30.9239	31.2981	31.0187	30.9662	30.9477	30.9391
5	40.2500	40.9058	40.415	40.3235	40.2913	40.2765
6	46.5976	47.72	46.9183	46.7409	46.6783	46.6493
7	46.9583	47.8482	47.1506	47.044	47.0065	46.9892
8	51.0891	52.4299	51.4271	51.2397	51.1739	51.1434
9	60.3750	62.5888	60.9322	60.623	60.5145	60.4643
10	61.8478	64.6263	62.592	62.1831	62.0373	61.9694
11	64.7797	68.0695	65.5868	65.1372	64.9806	64.9082
12	70.4375	73.0004	71.0859	70.7264	70.6002	70.5417

**Table 10 - Estimated eigenfunctions for the propagation of sound in a rectangular cavity using linear triangular elements with 50 linear divisions over  $(-12 < x < 12, -14 < y < 14)$ .**



## VII. CONCLUSION

I have reviewed the FEM as applied to some simple one- and two-dimensional eigenproblems. I developed an object-oriented computer code and utilized it to solve a number of one-dimensional systems. These included the isotropic harmonic oscillator, finite well, infinite well, and radial hydrogen atom. The finite element analysis of these systems provided very accurate results. The computer code was easily modified and applied to the one-dimensional unbound quantum mechanical system of a square barrier potential and also provided accurate results. I then developed a two-dimensional FEM code, and used this code to analyze the two-dimensional isotropic harmonic oscillator, and the propagational modes of sound in a rectangular cavity. Even with the use of simple elements this two-dimensional code also produced fairly accurate results.

## APPENDIX

### DATA INPUT INSTRUCTIONS AND LISTINGS FOR TISEFEA AND FEM2DLINER

## A. DATA INPUT INSTRUCTIONS FOR TISEFEA

Input into TISEFEA is fairly simple. The input has the following format:

```
NED NN DOF ZN
Z1
Z2
:
:
Zn
```

where

**NED** – Number of elements in the domain (integer)  
**NN** – Number of nodes per element (integer)  
**DOF** – Degrees of freedom per node (integer)  
**ZN** – Number of property zones (integer)  
**Z1, Z2, ..., Zn** – Property zones 1 through n  
Each property zone, **Zn**, has the following form:  
**NEZ ZLB ZRB A1 A2 A3 B1 B2 B3 G1 G2 G3**

where

**NEZ** – Number of elements in the zone (integer)  
**ZLB** – Zone left bound (real)  
**ZRB** – Zone right bound (real)  
**A1** – Constant term for Alpha (real) \  
**A2** – Linear term for Alpha (real) ----  $\text{Alpha}(x) = \mathbf{A1} + \mathbf{A2} * x + \mathbf{A3} * x * x$   
**A3** – Quadratic term for Alpha (real) /  
**B1** – Constant term for Beta (real) \  
**B2** – Linear term for Beta (real) ----  $\text{Beta}(x) = \mathbf{A1} + \mathbf{A2} * x + \mathbf{A3} * x * x$   
**B3** – Quadratic term for Beta (real) /  
**G1** – Constant term for Gamma (real) \  
**G2** – Linear term for Gamma (real) ----  $\text{Alpha}(x) = \mathbf{A1} + \mathbf{A2} * x + \mathbf{A3} * x * x$   
**G3** – Quadratic term for Gamma (real) /

*Example 1:* Harmonic oscillator with 10 elements of type 2-1 and a domain from –3 to 3:

```
10 2 1 1
10 -3 3 1 0 0 0 0 1 1 0 0
```

*Example 2:* Particle in a finite well of barrier width 6 and height 4, with a domain from –10 to 10. There are 100 equally spaced elements of type 2-3:

**100 2 3 3**  
**35 -10 -3 1 0 0 4 0 0 1 0 0**  
**30 -3 3 1 0 0 0 0 0 1 0 0**  
**35 3 10 1 0 0 4 0 0 1 0 0**

*Example 3:* Tunneling through a barrier of width 3 and height 4.25. There are 40 equally spaced elements of type 4-1. The particle energy is 2:

**40 4 1 1**  
**40 0 3 1 0 0 4.25 0 0 2 0 0**

## B. TISEFEA CODE LISTING

```
// *****
// TISEFEA.cpp
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Eigensolution implementation, main file
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#include <iostream>
#include <fstream>
#include "Model.h"

using namespace std;
using namespace TNT;

int main(int argc, char* argv[])
{
    ofstream outFile;

    Model TISE_Model;

    TISE_Model.GetModelData();

    TISE_Model.Solve();

    if(argc==2)
    {
        outFile.open(argv[1]);

        if(!outFile)
        {
            cout<<"Failed to open the output file: "<<argv[1]<<endl;
            exit(1);
        }
        else
        {
            cout<<"Writing to the output file: "<<argv[1]<<endl;

            TISE_Model.OutputSolution(outFile);
        }
    }
    else
    {
        cout<<"Program needs to recieve an output file as an argument."<<endl;
        exit(1);
    }
}
```



```

    outFile.close();

    return 0;
}
// end of TISEFEA.cpp

// *****
// Model.h
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Eigensolution implementation
//
// Interface for the Model class: Sets model data, assembles and
// solves for the approximate eigensolutions, and outputs
// solutions
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#ifndef MODEL_H
#define MODEL_H

#include <iostream>
#include <fstream>
#include <iomanip>
#include "Element.h"
#include "TNT\tnt.h"
#include "TNT\vec.h"
#include "TNT\cmat.h"
#include "TNT\fmt.h"
#include "TNT\transv.h"

using namespace std;
using namespace TNT;

class Model
{
public:
    .....Mo
    del();
    .....~M
    odel();
    void GetModelData();
    void Solve();
    void OutputSolution(ofstream&);

private:
    int numElements;
    int numNodesPerElement;

```

```

int numDofPerNode;
Element* elements;
Matrix<double> K;
Matrix<double> M;
Matrix<double> Eigenvecs;
Vector<double> Eigenvals;
void SolveGenSymEigen(const Matrix<double> &A, const Matrix<double> &B,
                     Matrix<double> &Eigenvectors, Vector<double> &Eigenvalues,
                     int &Info);

};

extern "C"
{
// solve general symmetric eigenvalues, eigenvectors
//
void dsygv_( const int *itype, char *jobz, char *uplo, const int *N,
             double *A, const int *lda, double *B, const int *ldb, double *W,
             double *work, const int *lwork, int *info);
}

#endif
// End of header file Model.h

// *****
// Model.cpp
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Eigensolution implementation
//
// Implementation of the Model class: Sets model data, assembles
// and solves for the approximate eigensolutions, and outputs
// solutions
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#include "Model.h"

Model::Model()
{

}

Model::~~Model()
{
    delete [] elements;
}

void Model::GetModelData()

```

```

{
    int numElemInZone = 0;
    int numZones = 0;
    double lBound = 0.0;
    double rBound = 0.0;
    double elLength = 0.0;
    double elLBound = 0.0;
    double elRBound = 0.0;
    QuadPolyCoeffs alpha;
    QuadPolyCoeffs beta;
    QuadPolyCoeffs gamma;

    cin>>numElements>>numNodesPerElement>>numDofPerNode>>numZones;

    elements = new Element[numElements];

    int el = 0;

    for(int zone = 0; zone<numZones; zone++)
    {
        cin >>numElemInZone>>lBound>>rBound>>alpha.a>>alpha.b>>alpha.c
            >>beta.a>>beta.b>>beta.c>>gamma.a>>gamma.b>>gamma.c;

        elLength = (rBound-lBound)/numElemInZone;

        if(!zone)
        {
            elLBound = lBound;
        }

        int elStartNum = el;
        for(;el<elStartNum+numElemInZone-1;el++)
        {
            elRBound = elLBound + elLength;
            elements[el].SetElementData(elLBound,elRBound,numNodesPerElement,
                numDofPerNode,alpha,beta,gamma);
            elLBound += elLength;
        }

        elements[el].SetElementData(elLBound,rBound,numNodesPerElement,
            numDofPerNode,alpha,beta,gamma);

        el++;
        elLBound = rBound;
    }
}

void Model::Solve()
{
    // assemble the elements
    for(int el=0;el<numElements;el++)
    {
        elements[el].AssembleElement();
    }
}

```

```

}

// reallocate the buffers
int elMatSize = numNodesPerElement*numDofPerNode;
int modMatSize = numElements*elMatSize-(numElements-1)*numDofPerNode;
int OK = 0;
K = K.newsize(modMatSize,modMatSize);
M = M.newsize(modMatSize,modMatSize);
Eigenvecs = Eigenvecs.newsize(modMatSize,modMatSize);
Eigenvals = Eigenvals.newsize(modMatSize);

// zero out the buffers
K=0.0;
M=0.0;
Eigenvecs=0.0;
Eigenvals=0.0;

Index1D I(1,elMatSize);

// temp matrices
Matrix<double> KTemp(modMatSize,modMatSize);
Matrix<double> MTemp(modMatSize,modMatSize);

// assemble the system equations
for(int i=0;i<numElements;i++)
{
    // zero out Temps
    KTemp = 0.0;
    MTemp = 0.0;

    KTemp(I+i*(elMatSize-numDofPerNode),I+i*(elMatSize-numDofPerNode))
        = elements[i].K(I,I);
    MTemp(I+i*(elMatSize-numDofPerNode),I+i*(elMatSize-numDofPerNode))
        = elements[i].M(I,I);

    K = K + KTemp;
    M = M + MTemp;
}

// solve the general symmetric eigenproblem
SolveGenSymEigen(K,M,Eigenvecs,Eigenvals,OK);

// check if we found a solution
if(OK!=0)
{
    cout<<"The eigenproblem solution failed with failure code: "<<OK<<endl<<endl;
}

}

// solve symmetric general eigenvalue problem
//
void Model::SolveGenSymEigen(const Matrix<double> &A, const Matrix<double> &B,

```

```

        Matrix<double> &Eigenvectors, Vector<double> &Eigenvalues,
        int &Info)
{
    assert(A.size() == B.size());
    assert(A.size() == Eigenvectors.size());

    Subscript N = A.num_rows();

    assert(N == A.num_cols());
    assert(N == Eigenvalues.size());

    char jobz = ' V' ;
    char uplo = ' U' ;
    int itype = 1;
    int worksize = 3*N;
    Vector<double> work(worksize);
    Fortran_Matrix<double> Tmp1( A.num_cols(), A.num_rows(), &A(1,1));
    Fortran_Matrix<double> Tmp2( B.num_cols(), B.num_rows(), &B(1,1));

    dsygv_(&itype, &jobz, &uplo, &N, &Tmp1(1,1), &N, &Tmp2(1,1), &N,
        Eigenvalues.begin(), work.begin(), &worksize, &Info);

    Matrix<double> Tmp3( Tmp1.num_cols(), Tmp1.num_rows(), &Tmp1(1,1));

    Eigenvectors = Tmp3;
}

void Model::OutputSolution(ofstream &outStream)
{
    outStream<<"  "<<"\t";
    for(int i = 0; i<numElements*numNodesPerElement
        *numDofPerNode-(numElements-1)*numDofPerNode && i<10; i++)
    {
        outStream<<setw(18)<<left<<setprecision(10)<<Eigenvals[i]<<"\t";
    }
    outStream<<endl<<endl;
    for(int j = 0; j<numElements*numNodesPerElement*numDofPerNode-(numElements-
1)*numDofPerNode; j++)
    {
        outStream<<setw(4)<<left<<uppercase<<j+1<<"\t"<<setw(18)<<Eigenvecs[0][j]<<"\t"
            <<setw(18)<<Eigenvecs[1][j]<<"\t"<<setw(18)<<Eigenvecs[2][j]<<"\t"
            <<setw(18)<<Eigenvecs[3][j]<<"\t"<<setw(18)<<Eigenvecs[4][j]<<"\t"<<endl;
    }
}
// end of Model.cpp

// *****
// Element.h
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Eigensolution implementation
//
// Interface for the Element class: Sets element data and

```

```

// performs Gauss-Legendre quadrature to fill element equations
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#ifndef ELEMENT_H
#define ELEMENT_H

#include <iostream>
#include <cmath>
#include "Shapes.h"
#include "TNT\tnt.h"
#include "TNT\cmat.h"

using namespace std;
using namespace TNT;

const int numGaussPoints = 12;

struct QuadPolyCoeffs
{
    // = a + b*x + c*x*x
    double a;
    double b;
    double c;
};

class Element
{
public:
    .....Ele
    ment();
    .....~El
    ement();
    void SetElementData(const double &leftBound, const double &rightBound,
                        const int &numberNodesPerElement, const int &numberDofPerNode,
                        const QuadPolyCoeffs &alpha, const QuadPolyCoeffs &beta,
                        const QuadPolyCoeffs &gamma);
    double LeftBound();
    double RightBound();
    double Length();
    int NumberOfNodes();
    int DofPerNode();
    void AssembleElement();
    Matrix<double> K;
    Matrix<double> M;

private:
    double GLIntegration(int &Row, int &Col, double (Element::*IntegrandFcn)(int&, int&, double&));
    double AlphaIntegrand(int &shapeNumRow, int &shapeNumCol, double &Xi);

```

```

double BetaIntegrand(int &shapeNumRow, int &shapeNumCol, double &Xi);
double GammaIntegrand(int &shapeNumRow, int &shapeNumCol, double &Xi);
double Alpha(const double &x);
double Beta(const double &x);
double Gamma(const double &x);
QuadPolyCoeffs alphaCoeffs;
QuadPolyCoeffs betaCoeffs;
QuadPolyCoeffs gammaCoeffs;
Shapes shapeFcns;
double gaussPoints[numGaussPoints];
double weights[numGaussPoints];
double lBound;
double rBound;
double length;
int numNodes;
int numDof;
};

#endif
// End of header file Element.h

// *****
// Element.cpp
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Eigensolution implementation
//
// Implementation of the Element class: Sets element data and
// performs Gauss-Legendre quadrature to fill element equations
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#include "Element.h"

Element::Element()
{
    gaussPoints[0] = -0.981560634246719;
    gaussPoints[1] = -0.904117256370475;
    gaussPoints[2] = -0.769902674194305;
    gaussPoints[3] = -0.587317954286617;
    gaussPoints[4] = -0.367831498998180;
    gaussPoints[5] = -0.125233408511469;
    gaussPoints[6] = 0.125233408511469;
    gaussPoints[7] = 0.367831498998180;
    gaussPoints[8] = 0.587317954286617;
    gaussPoints[9] = 0.769902674194305;
    gaussPoints[10] = 0.904117256370475;
    gaussPoints[11] = 0.981560634246719;

```

```

weights[0] = 0.047175336386512;
weights[1] = 0.106939325995318;
weights[2] = 0.160078328543346;
weights[3] = 0.203167426723066;
weights[4] = 0.233492536538355;
weights[5] = 0.249147045813403;
weights[6] = 0.249147045813403;
weights[7] = 0.233492536538355;
weights[8] = 0.203167426723066;
weights[9] = 0.160078328543346;
weights[10] = 0.106939325995318;
weights[11] = 0.047175336386512;
}

Element::~Element()
{

}

void Element::SetElementData(const double &leftBound, const double &rightBound,
                             const int &numberNodesPerElement, const int &numberDofPerNode,
                             const QuadPolyCoeffs &alpha, const QuadPolyCoeffs &beta,
                             const QuadPolyCoeffs &gamma)
{
    lBound = leftBound;
    rBound = rightBound;
    length = rBound-lBound;
    numNodes = numberNodesPerElement;
    numDof = numberDofPerNode;
    alphaCoeffs = alpha;
    betaCoeffs = beta;
    gammaCoeffs = gamma;

    shapeFcns.SetShapes(numNodes, numDof, lBound, length);

    // size the element equation matrices
    K = K.newsize(numNodes*numDof,numNodes*numDof);
    K=0.0;
    M = M.newsize(numNodes*numDof,numNodes*numDof);
    M=0.0;
}

double Element::LeftBound()
{
    return lBound;
}

double Element::RightBound()
{
    return rBound;
}

double Element::Length()
{

```



```

    return length;
}

int Element::NumberOfNodes()
{
    return numNodes;
}

int Element::DofPerNode()
{
    return numDof;
}

void Element::AssembleElement()
{
    for(int i = 1; i <= numNodes*numDof; i++)
    {
        for(int j = 1; j <= numNodes*numDof; j++)
        {
            K[i-1][j-1] = GLIntegration(i, j, AlphaIntegrand) + GLIntegration(i, j, BetaIntegrand);
            M[i-1][j-1] = GLIntegration(i, j, GammaIntegrand);
        }
    }
}

double Element::Alpha(const double &x)
{
    return alphaCoeffs.a+alphaCoeffs.b*x+alphaCoeffs.c*x*x;
}

double Element::Beta(const double &x)
{
    return betaCoeffs.a+betaCoeffs.b*x+betaCoeffs.c*x*x;
}

double Element::Gamma(const double &x)
{
    return gammaCoeffs.a+gammaCoeffs.b*x+gammaCoeffs.c*x*x;
}

double Element::GLIntegration(int &Row, int &Col, double (Element::*IntegrandFcn)(int&, int&, double&))
{
    double Sum = 0.0;
    for(int i = 0; i < numGaussPoints; i++)
        Sum += weights[i]*(this->*IntegrandFcn)(Row, Col, gaussPoints[i]);

    return Sum;
}

double Element::AlphaIntegrand(int &shapeNumRow, int &shapeNumCol, double &Xi)
{
    return shapeFcns.DPhi(shapeNumRow, Xi)*Alpha(shapeFcns.X(Xi))
        *shapeFcns.DPhi(shapeNumCol, Xi)/shapeFcns.Jacobian();
}

```

```

}

double Element::BetaIntegrand(int &shapeNumRow, int &shapeNumCol, double &Xi)
{
    return shapeFcns.Phi(shapeNumRow, Xi)*Beta(shapeFcns.X(Xi))
           *shapeFcns.Phi(shapeNumCol, Xi)*shapeFcns.Jacobian();
}

double Element::GammaIntegrand(int &shapeNumRow, int &shapeNumCol, double &Xi)
{
    return shapeFcns.Phi(shapeNumRow, Xi)*Gamma(shapeFcns.X(Xi))
           *shapeFcns.Phi(shapeNumCol, Xi)*shapeFcns.Jacobian();
}
// end of Element.cpp

// *****
// Shapes.h
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Interface for the Shapes class: Collection of all the
// various shape functions, their derivatives, the Jacobian,
// and the coordinate transformation function.
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#ifndef SHAPES_H
#define SHAPES_H

#include <iostream>

using namespace std;

class Shapes
{
public:
    .....Sha
pes();
    .....~Sh
apes();
    void SetShapes(const int &numberNodesPerElement, const int &numberDofPerNode,
                  const double &elementLeftBound, const double &elementLength);
    double X(const double &Xi);
    double Phi(const int &shapeNum, const double &Xi);
    double DPhi(const int &shapeNum, const double &Xi);
    double Jacobian();

private:
    int numNodes;
    int numDof;

```

```

    double elLeftBound;
    double elLength;
};

#endif
// End of header file Shapes.h

// *****
// Shapes.cpp
//
// Time-Independent, Shroedinger Equation Finite Element Analysis
//
// Implementation of the Shapes class: Collection of all the
// various shape functions, their derivatives, the Jacobian,
// and the coordinate transformation function.
//
// Jason C. Hunnell
//
// August 8, 2001
//
// *****

#include "Shapes.h"

Shapes::Shapes()
{

}

Shapes::~Shapes()
{

}

void Shapes::SetShapes(const int &numberNodesPerElement, const int &numberDofPerNode,
                      const double &elementLeftBound, const double &elementLength)
{
    numNodes = numberNodesPerElement;
    numDof = numberDofPerNode;
    elLeftBound = elementLeftBound;
    elLength = elementLength;
}

double Shapes::Jacobian()
{
    {
        return elLength/2.0;
    }
}

double Shapes::Phi(const int &shapeNum, const double &Xi)
{
    {
        switch(numNodes)
        {
            case 2:
                switch(numDof)

```

```

{
case 1:
switch(shapeNum)
{
case 1:
return (0.5)*(1.0-Xi);
case 2:
return (0.5)*(1.0+Xi);
default:
cout<<"Error in Shape::Phi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
case 2:
switch(shapeNum)
{
case 1:
return (0.25)*(Xi*Xi*Xi-3.0*Xi+2.0);
case 2:
return (0.25)*(Xi-1.0)*(Xi-1.0)*(Xi+1.0)*Jacobian();
case 3:
return (-0.25)*(Xi-2.0)*(Xi+1.0)*(Xi+1.0);
case 4:
return (0.25)*(Xi-1.0)*(Xi+1.0)*(Xi+1.0)*Jacobian();
default:
cout<<"Error in Shape::Phi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
case 3:
switch(shapeNum)
{
case 1:
return (-0.0625)*(Xi-1.0)*(Xi-1.0)*(Xi-1.0)*(3*Xi*Xi+9.0*Xi+8.0);
case 2:
return (-0.0625)*(Xi-1.0)*(Xi-1.0)*(Xi-1.0)*(Xi+1.0)*(3.0*Xi+5.0)*Jacobian();
case 3:
return (-0.0625)*(Xi-1.0)*(Xi-1.0)*(Xi-1.0)*(Xi+1.0)*(Xi+1.0)*Jacobian()*Jacobian();
case 4:
return (0.0625)*(Xi+1.0)*(Xi+1.0)*(Xi+1.0)*(3*Xi*Xi-9.0*Xi+8.0);
case 5:
return (-0.0625)*(Xi-1.0)*(Xi+1.0)*(Xi+1.0)*(Xi+1.0)*(3.0*Xi-5.0)*Jacobian();
case 6:
return (0.0625)*(Xi-1.0)*(Xi-1.0)*(Xi+1.0)*(Xi+1.0)*(Xi+1.0)*Jacobian()*Jacobian();
default:
cout<<"Error in Shape::Phi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
default:
cout<<"Error in Shape::Phi : No case data for numberDofPerNode = "

```

```

        <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
        exit(1);
    }
case 3:
    switch(numDof)
    {
    case 1:
        switch(shapeNum)
        {
        case 1:
            return (1.0/2.0)*Xi*(Xi-1.0);
        case 2:
            return (1.0+Xi)*(1.0-Xi);
        case 3:
            return (1.0/2.0)*Xi*(Xi+1.0);
        default:
            cout<<"Error in Shape::Phi : No case data for shapNum = "
                <<shapeNum<<", numberDofPerNode = "<<numDof
                <<", and numberNodesPerElement = "<<numNodes<<endl;
            exit(1);
        }
    case 2:
        switch(shapeNum)
        {
        case 1:
            return 0.25*(4.0*Xi*Xi-5.0*Xi*Xi*Xi-2.0*Xi*Xi*Xi*Xi+3.0*Xi*Xi*Xi*Xi*Xi);
        case 2:
            return 0.25*(Xi*Xi-Xi*Xi*Xi-Xi*Xi*Xi*Xi+Xi*Xi*Xi*Xi*Xi)*Jacobian();
        case 3:
            return 1.0-2.0*Xi*Xi+Xi*Xi*Xi*Xi;
        case 4:
            return (Xi-2.0*Xi*Xi*Xi+Xi*Xi*Xi*Xi*Xi)*Jacobian();
        case 5:
            return 0.25*(4.0*Xi*Xi+5.0*Xi*Xi*Xi-2.0*Xi*Xi*Xi*Xi-3.0*Xi*Xi*Xi*Xi*Xi);
        case 6:
            return 0.25*(-Xi*Xi-Xi*Xi*Xi+Xi*Xi*Xi*Xi+Xi*Xi*Xi*Xi*Xi)*Jacobian();
        default:
            cout<<"Error in Shape::Phi : No case data for shapNum = "
                <<shapeNum<<", numberDofPerNode = "<<numDof
                <<", and numberNodesPerElement = "<<numNodes<<endl;
            exit(1);
        }
    default:
        cout<<"Error in Shape::Phi : No case data for numberDofPerNode = "
            <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
        exit(1);
    }
case 4:
    switch(numDof)
    {
    case 1:
        switch(shapeNum)
        {
        case 1:

```

```

        return (-9./16.)*(Xi+1./3.)*(Xi-1./3.)*(Xi-1.);
    case 2:
        return (27./16.)*(Xi+1.)*(Xi-1./3.)*(Xi-1.);
    case 3:
        return (-27./16.)*(Xi+1.)*(Xi+1./3.)*(Xi-1.);
    case 4:
        return (9./16.)*(Xi+1.)*(Xi+1./3.)*(Xi-1./3.);
    default:
        cout<<"Error in Shape::Phi : No case data for shapNum = "
            <<shapeNum<<" , numberDofPerNode = "<<numDof
            <<" , and numberNodesPerElement = "<<numNodes<<endl;
        exit(1);
    }
default:
    cout<<"Error in Shape::Phi : No case data for numberDofPerNode = "
        <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
    exit(1);
}
case 5:
    switch(numDof)
    {
    case 1:
        switch(shapeNum)
        {
        case 1:
            return (2./3.)*(Xi+.5)*Xi*(Xi-.5)*(Xi-1.);
        case 2:
            return (-8./3.)*(Xi+1.)*Xi*(Xi-.5)*(Xi-1.);
        case 3:
            return 4.*(Xi+1.)*(Xi+.5)*(Xi-.5)*(Xi-1.);
        case 4:
            return (-8./3.)*(Xi+1.)*Xi*(Xi+.5)*(Xi-1.);
        case 5:
            return (2./3.)*(Xi+.5)*Xi*(Xi-.5)*(Xi+1.);
        default:
            cout<<"Error in Shape::Phi : No case data for shapNum = "
                <<shapeNum<<" , numberDofPerNode = "<<numDof
                <<" , and numberNodesPerElement = "<<numNodes<<endl;
            exit(1);
        }
    default:
        cout<<"Error in Shape::Phi : No case data for numberDofPerNode = "
            <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
        exit(1);
    }
default:
    cout<<"Error in Shape::Phi : No case data for numberNodesPerElement = "
        <<numNodes<<endl;
    exit(1);
}
}
}

double Shapes::DPhi(const int &shapeNum, const double &Xi)
{

```

```

switch(numNodes)
{
case 2:
switch(numDof)
{
case 1:
switch(shapeNum)
{
case 1:
return -0.5;
case 2:
return 0.5;
default:
cout<<"Error in Shape::DPhi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
}
case 2:
switch(shapeNum)
{
case 1:
return (0.75)*(Xi*Xi-1.0);
case 2:
return (0.25)*(3.0*Xi*Xi-2.0*Xi-1.0)*Jacobian();
case 3:
return (-0.75)*(Xi*Xi-1.0);
case 4:
return (0.25)*(3.0*Xi*Xi+2.0*Xi-1.0)*Jacobian();
default:
cout<<"Error in Shape::DPhi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
}
case 3:
switch(shapeNum)
{
case 1:
return (-0.9375)*(Xi*Xi-1.0)*(Xi*Xi-1.0);
case 2:
return (-0.0625)*(Xi-1.0)*(Xi-1.0)*(15.0*Xi*Xi+26.0*Xi+7.0)*Jacobian();
case 3:
return (-0.0625)*(Xi-1.0)*(Xi-1.0)*(5.0*Xi*Xi+6.0*Xi+1.0)*Jacobian()*Jacobian();
case 4:
return (0.9375)*(Xi*Xi-1.0)*(Xi*Xi-1.0);
case 5:
return (-0.0625)*(Xi+1.0)*(Xi+1.0)*(15.0*Xi*Xi-26.0*Xi+7.0)*Jacobian();
case 6:
return (0.0625)*(Xi+1.0)*(Xi+1.0)*(5.0*Xi*Xi-6.0*Xi+1.0)*Jacobian()*Jacobian();
default:
cout<<"Error in Shape::DPhi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;

```

```

        exit(1);
    }
default:
    cout<<"Error in Shape::DPhi : No case data for numberDofPerNode = "
        <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
    exit(1);
}
case 3:
    switch(numDof)
    {
    case 1:
        switch(shapeNum)
        {
        case 1:
            return Xi-0.5;
        case 2:
            return -2.0*Xi;
        case 3:
            return Xi+0.5;
        default:
            cout<<"Error in Shape::DPhi : No case data for shapNum = "
                <<shapeNum<<", numberDofPerNode = "<<numDof
                <<", and numberNodesPerElement = "<<numNodes<<endl;
            exit(1);
        }
    case 2:
        switch(shapeNum)
        {
        case 1:
            return 0.25*(8.0*Xi-15.0*Xi*Xi-8.0*Xi*Xi*Xi+15.0*Xi*Xi*Xi*Xi);
        case 2:
            return 0.25*(2.0*Xi-3.0*Xi*Xi-4.0*Xi*Xi*Xi+5.0*Xi*Xi*Xi*Xi)*Jacobian();
        case 3:
            return -4.0*Xi+4.0*Xi*Xi*Xi;
        case 4:
            return (1.0-6.0*Xi*Xi+5.0*Xi*Xi*Xi*Xi)*Jacobian();
        case 5:
            return 0.25*(8.0*Xi+15.0*Xi*Xi-8.0*Xi*Xi*Xi-15.0*Xi*Xi*Xi*Xi);
        case 6:
            return 0.25*(-2.0*Xi-3.0*Xi*Xi+4.0*Xi*Xi*Xi+5.0*Xi*Xi*Xi*Xi)*Jacobian();
        default:
            cout<<"Error in Shape::DPhi : No case data for shapNum = "
                <<shapeNum<<", numberDofPerNode = "<<numDof
                <<", and numberNodesPerElement = "<<numNodes<<endl;
            exit(1);
        }
    default:
        cout<<"Error in Shape::DPhi : No case data for numberDofPerNode = "
            <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
        exit(1);
    }
}
case 4:
    switch(numDof)
    {

```



```

case 1:
switch(shapeNum)
{
case 1:
return (1./16.)*(1.+18.*Xi-27.*Xi*Xi);
case 2:
return (9./16.)*(-3.-2.*Xi+9.*Xi*Xi);
case 3:
return -(9./16.)*(-3.+2.*Xi+9.*Xi*Xi);
case 4:
return (1./16.)*(-1.+18.*Xi+27.*Xi*Xi);
default:
cout<<"Error in Shape::DPhi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
default:
cout<<"Error in Shape::DPhi : No case data for numberDofPerNode = "
<<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
case 5:
switch(numDof)
{
case 1:
switch(shapeNum)
{
case 1:
return (1./6.)*(1. - 2.*Xi - 12.*Xi*Xi + 16.*Xi*Xi*Xi);
case 2:
return -(4./3.)*(1. - 4.*Xi - 3.*Xi*Xi + 8.*Xi*Xi*Xi);
case 3:
return 2.*Xi*(-5. + 8.*Xi*Xi);
case 4:
return -(4./3.)*(-1. - 4.*Xi + 3.*Xi*Xi + 8.*Xi*Xi*Xi);
case 5:
return (1./6.)*(-1. - 2.*Xi + 12.*Xi*Xi + 16.*Xi*Xi*Xi);
default:
cout<<"Error in Shape::DPhi : No case data for shapNum = "
<<shapeNum<<", numberDofPerNode = "<<numDof
<<", and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
default:
cout<<"Error in Shape::DPhi : No case data for numberDofPerNode = "
<<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
default:
cout<<"Error in Shape::DPhi : No case data for numberNodesPerElement = "
<<numNodes<<endl;
exit(1);
}
}

```

```

}

double Shapes::X(const double &Xi)
{
    switch(numNodes)
    {
        case 2:
            switch(numDof)
            {
                case 1:
                    return elLeftBound*(0.5)*(1.0-Xi)+(elLeftBound+elLength)*(0.5)*(1.0+Xi);
                case 2:
                    return elLeftBound*(0.5)*(1.0-Xi)+(elLeftBound+elLength)*(0.5)*(1.0+Xi);
                // return 0.25*(elLeftBound*(Xi-1.0)*(Xi-1.0)*(2.0+Jacobian()+Xi+Jacobian()*Xi)
                // + (elLeftBound+elLength)*(1.0+Xi)*(1.0+Xi)*(2.+Jacobian()*(Xi-1.)-Xi));
                case 3:
                    return (0.5)*(elLeftBound*(1.-Xi)+(elLeftBound+elLength)*(1.+Xi));
                // return 0.0625*(-elLeftBound*(Xi-1.0)*(Xi-1.0)*(Xi-1.0)*(8.0+9.0*Xi+3.0*Xi*Xi
                // +Jacobian()*Jacobian()*(Xi+1.0)*(Xi+1.0)+Jacobian()*(5.0+8.0*Xi+3.0*Xi*Xi))
                // + (elLeftBound+elLength)*(1.0+Xi)*(1.0+Xi)*(1.0+Xi)*(8.0+Jacobian()
                // *Jacobian()*(Xi-1.0)*(Xi-1.0)-9.0*Xi+3.0*Xi*Xi+Jacobian()*(-5.0+8.0*Xi
                // -3.0*Xi*Xi));
                default:
                    cout<<"Error in Shape::X : No case data for numberDofPerNode = "
                     <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
                    exit(1);
            }
        case 3:
            switch(numDof)
            {
                case 1:
                    return (0.5)*(1.0-Xi)*elLeftBound + (0.5)*(1.0+Xi)*(elLeftBound+elLength);
                case 2:
                    return 0.25*(elLeftBound*(3.*(Jacobian()+1.)*Xi*Xi*Xi+(5.*(Jacobian()+6.)*Xi*Xi+2.
                    *(Jacobian()+2.)*Xi+2.)*(Xi-1.)*(Xi-1.)+(elLeftBound+elLength)*(Xi+1.)
                    *(Xi+1.)*(3.*(Jacobian()-1.)*Xi*Xi*Xi+(6.-5.*(Jacobian()))*Xi*Xi+2.
                    *(Jacobian()-2.)*Xi+2.));
                default:
                    cout<<"Error in Shape::X : No case data for numberDofPerNode = "
                     <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
                    exit(1);
            }
        case 4:
            switch(numDof)
            {
                case 1:
                    return (0.5)*(elLeftBound*(1.-Xi)+(elLeftBound+elLength)*(1.+Xi));
                default:
                    cout<<"Error in Shape::X : No case data for numberDofPerNode = "
                     <<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
                    exit(1);
            }
        case 5:
            switch(numDof)

```

```

{
case 1:
return (0.5)*(elLeftBound*(1.-Xi)+(elLeftBound+elLength)*(1.+Xi));
default:
cout<<"Error in Shape::X : No case data for numberDofPerNode = "
<<numDof<<" and numberNodesPerElement = "<<numNodes<<endl;
exit(1);
}
default:
cout<<"Error in Shape::X : No case data for numberNodesPerElement = "
<<numNodes<<endl;
exit(1);
}
}
// end of Shapes.cpp

```

### C. FEM2DLINER CODE LISTING

```

// *****
// FEM2DLinear.cpp
//
// Finite Element Method with Two-Dimensional Linear Triangles
//
// Eigensolution implementation, main file
//
// Jason C. Hunnell
//
// March 8, 2002
//
// *****

#include <iostream>
#include <fstream>
#include "FEM2DLinear.h"

using namespace std;
using namespace TNT;

ofstream outFile;

int main(int argc, char* argv[])
{
if(argc==2)
{
outFile.open(argv[1]);

if(!outFile)
{
cout<<"Failed to open the output file: "<<argv[1]<<endl;
exit(1);
}
}
else
{

```

```

    cout<<"Program needs to recieve an output file as an argument."<<endl;
    exit(1);
}

SetModelData();

Solve();

cout<<"Writing to the output file: "<<argv[1]<<endl;

OutputSolution(outFile);

DisplayMain(argc, argv);

outFile.close();

CleanUp();

return 0;
}

void SetModelData()
{
    QuadPolyCoeffs shoAtts[] = {
        {1.0,0.0,0.0,0.0,0.0,0.0,0.0},
        {1.0,0.0,0.0,0.0,0.0,0.0,0.0},
        {0.0,0.0,0.0,1.0,0.0,1.0},
        {1.0,0.0,0.0,0.0,0.0,0.0}
    };

    QuadPolyCoeffs acousticAtts[] = {
        {1.0,0.0,0.0,0.0,0.0,0.0,0.0},
        {1.0,0.0,0.0,0.0,0.0,0.0,0.0},
        {0.0,0.0,0.0,0.0,0.0,0.0},
        {1.0,0.0,0.0,0.0,0.0,0.0}
    };

    attributes[0] = acousticAtts[0];
    attributes[1] = acousticAtts[1];
    attributes[2] = acousticAtts[2];
    attributes[3] = acousticAtts[3];

    Lx = 24;
    Ly = 28;
    Ix = 10;
    Iy = 10;

    numOfNodes = (Ix+1)*(Iy+1);
    numOfTriangles = 2*Ix*Iy;

    nodeCoords = new double[numOfNodes][3]; // array of node coordinates
    // array of triangle nodes and attribute indices
    // for each triangle, {n1,n2,n3,n4,n5,n6,att}
    triangles = new int[numOfTriangles][3];
    vertNorms = new double[numOfNodes][3]; // array of vertex normal vectors

```

```

triNorms = new double[numOfTriangles][3]; // array of triangle normal vectors

FillNodeCoords();
FillTriangles();
ConstructVertexNormals();
}

void FillNodeCoords(void)
{
    double h = Lx/(double)Ix;
    double d = Ly/(double)Iy;
    double y = 0.0;
    double x = 0.0;
    int iNode = 0;
    for (int j = 0; j <= Iy; j++)
    {
        y = d*(double)j - (double)Ly/2.0;
        for (int i = 0; i <= Ix; i++)
        {
            x = h*(double)i - (double)Lx/2.0;
            nodeCoords[iNode][0] = x;
            nodeCoords[iNode][1] = y;
            iNode++;
        }
    }
}

void FillTriangles(void)
{
    int iNode = 0;
    int iTriangle = 0;
    for (int j = 1; j <= Iy; j++)
    {
        for (int i = 1; i <= Ix; i++)
        {
            iNode = j*(Ix+1)+i;
            triangles[iTriangle][0] = iNode;
            triangles[iTriangle][1] = iNode-1;
            triangles[iTriangle][2] = iNode-Ix-2;
            triangles[iTriangle+1][0] = iNode-Ix-2;
            triangles[iTriangle+1][1] = iNode-Ix-1;
            triangles[iTriangle+1][2] = iNode;
            iTriangle = iTriangle + 2;
        }
    }
}

void Solve()
{
    // reallocate the buffers
    K = K.newsize(numOfNodes,numOfNodes);
    M = M.newsize(numOfNodes,numOfNodes);
    Eigenvecs = Eigenvecs.newsize(numOfNodes,numOfNodes);
}

```

```

Eigenvals = Eigenvals.newsize(numOfNodes);

// zero out the buffers
K = 0.0;
M = 0.0;
Eigenvecs = 0.0;
Eigenvals = 0.0;

// assemble the system equations
for(int tri=0;tri<numOfTriangles;tri++)
{
    AssembleLinearElement(tri, K, M);
}

int OK = 0;

// solve the general symmetric eigenproblem
SolveGenSymEigen(K,M,Eigenvecs,Eigenvals,OK);

// check if we found a solution
if(OK!=0)
{
    cout<<"The eigenproblem solution failed with failure code: "<<OK<<endl<<endl;
}
}

void CleanUp()
{
    delete [] nodeCoords;
    delete [] triangles;

    nodeCoords = 0;
    triangles = 0;
}

// solve symmetric general eigenvalue problem
void SolveGenSymEigen(const Matrix<double> &A, const Matrix<double> &B,
                    Matrix<double> &Eigenvectors, Vector<double> &Eigenvalues,
                    int &Info)
{
    assert(A.size() == B.size());
    assert(A.size() == Eigenvectors.size());

    Subscript N = A.num_rows();

    assert(N == A.num_cols());
    assert(N == Eigenvalues.size());

    char jobz = ' V' ;
    char uplo = ' U' ;
    int itype = 1;
    int worksize = 3*N;
    Vector<double> work(worksize);
    Fortran_Matrix<double> Tmp1( A.num_cols(), A.num_rows(), &A(1,1));

```

```

Fortran_Matrix<double> Tmp2( B.num_cols(), B.num_rows(), &B(1,1));

dsygv_(&itype, &jobz, &uplo, &N, &Tmp1(1,1), &N, &Tmp2(1,1), &N,
Eigenvalues.begin(), work.begin(), &worksize, &Info);

Matrix<double> Tmp3( Tmp1.num_cols(), Tmp1.num_rows(), &Tmp1(1,1));

Eigenvectors = Tmp3;
}

void OutputSolution(ofstream &outStream)
{
for(int i = 0; i<numOfNodes && i<25; i++)
{
outStream<<setw(10)<<left<<setprecision(6)<<sqrt(Eigenvals[i])*1127.0/(2.0*acos(-1))<<endl;
// outStream<<setw(10)<<left<<setprecision(6)<<Eigenvals[i]<<endl;
}
}

void AssembleLinearElement(int triNum, Matrix<double> &K, Matrix<double> &M)
{
double b[3] = {0};
double c[3] = {0};
double area = 0;
double xCentroid = 0;
double yCentroid = 0;
double factor = 0;

b[0] = nodeCoords[triangles[triNum][1]][1] - nodeCoords[triangles[triNum][2]][1];
b[1] = nodeCoords[triangles[triNum][2]][1] - nodeCoords[triangles[triNum][0]][1];
b[2] = nodeCoords[triangles[triNum][0]][1] - nodeCoords[triangles[triNum][1]][1];

c[0] = nodeCoords[triangles[triNum][2]][0] - nodeCoords[triangles[triNum][1]][0];
c[1] = nodeCoords[triangles[triNum][0]][0] - nodeCoords[triangles[triNum][2]][0];
c[2] = nodeCoords[triangles[triNum][1]][0] - nodeCoords[triangles[triNum][0]][0];

area = (b[1]*c[2]-b[2]*c[1])/2.0;

xCentroid = (nodeCoords[triangles[triNum][0]][0]
+ nodeCoords[triangles[triNum][1]][0]
+ nodeCoords[triangles[triNum][2]][0])/3.0;
yCentroid = (nodeCoords[triangles[triNum][0]][1]
+ nodeCoords[triangles[triNum][1]][1]
+ nodeCoords[triangles[triNum][2]][1])/3.0;

for(int i = 0; i < 3; i++)
{
for(int j = 0; j < 3; j++)
{
if (i==j) factor = 6.0;
if (i!=j) factor = 12.0;
K[triangles[triNum][i]][triangles[triNum][j]]
= K[triangles[triNum][i]][triangles[triNum][j]]
+ (b[i]*AlphaX(triNum, xCentroid, yCentroid)*b[j]
+ c[i]*AlphaY(triNum, xCentroid, yCentroid)*c[j])/(area*4.0)

```

```

        + Beta(triNum, xCentroid, yCentroid)*area/factor;

    M[triangles[triNum][i]][triangles[triNum][j]]
        = M[triangles[triNum][i]][triangles[triNum][j]]
        + Gamma(triNum, xCentroid, yCentroid)*area/factor;
    }
}
}

double AlphaX(const int &triNum, const double &x, const double &y)
{
    return attributes[0].c + attributes[0].x*x
        + attributes[0].y*y + attributes[0].xx*x*x
        + attributes[0].xy*x*y + attributes[0].yy*y*y;
}

double AlphaY(const int &triNum, const double &x, const double &y)
{
    return attributes[1].c + attributes[1].x*x
        + attributes[1].y*y + attributes[1].xx*x*x
        + attributes[1].xy*x*y + attributes[1].yy*y*y;
}

double Beta(const int &triNum, const double &x, const double &y)
{
    return attributes[2].c + attributes[2].x*x
        + attributes[2].y*y + attributes[2].xx*x*x
        + attributes[2].xy*x*y + attributes[2].yy*y*y;
}

double Gamma(const int &triNum, const double &x, const double &y)
{
    return attributes[3].c + attributes[3].x*x
        + attributes[3].y*y + attributes[3].xx*x*x
        + attributes[3].xy*x*y + attributes[3].yy*y*y;
}

void FillNodeZCoords()
{
    double value;

    for (int iNode = 0; iNode < numOfNodes; iNode++)
    {
        value = scaleFactor*Eigenvecs[eigNum][iNode];
        nodeCoords[iNode][2] = value;

        if (bbzSet)
        {
            if (value > bbzmax) bbzmax = value;
            if (value < bbzmin) bbzmin = value;
        }
        else
        {
            bbzSet = 1;

```



```

        bbzmax = value;
        bbzmin = value;
    }
}

void init(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);

    glLightModelf(GL_LIGHT_MODEL_LOCAL_VIEWER, 1.0);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glEnable(GL_DEPTH_TEST);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glVertexPointer(3, GL_DOUBLE, 0, nodeCoords);
    glNormalPointer(GL_DOUBLE, 0, vertNorms);
}

void display(void)
{
    GLfloat mat_specular[] = { 0.6, 0.6, 0.6, 1.0 };
    GLfloat mat_ambient[] = { 1.0, 0.0, 0.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    GLfloat light_ambient[] = { 0.3, 0.3, 0.5, 0.0 };

    glShadeModel(GL_SMOOTH);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);

    glPushMatrix();

    glRotatef(xrot, 1.0, 0.0, 0.0);
    glRotatef(yrot, 0.0, 1.0, 0.0);

    glRotatef(zrot, 0.0, 0.0, 1.0);

    glPushMatrix();
    glTranslatef(0.0, 0.0, (bbzmax-bbzmin)/2.0+bbzmin);
    glPushMatrix();
    glScalef(Lx, Ly, bbzmax-bbzmin);
    glutWireCube(1.0);
    glPopMatrix();
    glPopMatrix();

    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);

```

```

if (polyFillMode == GL_FILL)
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}
else
{
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
}

glDrawElements(GL_TRIANGLES, 3*numOfTriangles, GL_UNSIGNED_INT, triangles);

glPopMatrix();

glutSwapBuffers();
}

void reshape (int w, int h)
{
    if (w<=h)
    {
        // window width is less than height
        glViewport(0, (GLint)(h-w)/2, (GLint)w, (GLint)w);
    }
    else
    {
        // window width is greater than height
        glViewport((GLint)(w-h)/2, 0, (GLint)h, (GLint)h);
    }

    // set viewing values
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    radius = sqrt((bbzmax-bbzmin)*(bbzmax-bbzmin)+Lx*Lx +Ly*Ly)/2.0;

    glFrustum(-zoomFactor*radius, zoomFactor*radius,
              -zoomFactor*radius, zoomFactor*radius,
              1.5*radius, 4.5*radius);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef (0.0, 0.0, -3.0*radius);
}

void specialKey(int key, int x, int y)
{
    switch (key)
    {
        case GLUT_KEY_PAGE_UP:
            zoomFactor = 1.1*zoomFactor;
            reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
            glutPostRedisplay();
            break;
    }
}

```

```

case GLUT_KEY_PAGE_DOWN:
    zoomFactor = 0.9*zoomFactor;
    if (zoomFactor < 0.000005)
    {
        zoomFactor = 0.000005;
    }
    reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
    glutPostRedisplay();
    break;

default:
    break;
}
}

void key(unsigned char key, int x, int y)
{
    menu((int) key);
}

void makeMenu(void)
{
    glutCreateMenu(menu);

    glutAddMenuEntry("", 0);
    glutAddMenuEntry(" x: Increase x-axis rotation angle", ' x' );
    glutAddMenuEntry(" X: Decrease x-axis rotation angle", ' X' );
    glutAddMenuEntry(" y: Increase y-axis rotation angle", ' y' );
    glutAddMenuEntry(" Y: Decrease y-axis rotation angle", ' Y' );
    glutAddMenuEntry(" z: Increase z-axis rotation angle", ' z' );
    glutAddMenuEntry(" Z: Decrease z-axis rotation angle", ' Z' );
    glutAddMenuEntry(" b: Toggle display of bounding box", ' b' );
    glutAddMenuEntry(" d: Half the function values", ' d' );
    glutAddMenuEntry(" D: Double the function values", ' D' );
    glutAddMenuEntry(" f: Toggle polygons, filled/wireframe", ' f' );
    glutAddMenuEntry(" r: Restore default for all options", ' r' );
    glutAddMenuEntry("", 0);
    glutAddMenuEntry(" Terminate the program", 27);

    glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void menu(int item)
{
    switch (item)
    {
        case ' x' :
            xrot += 2.0;
            if (xrot > 360.0) xrot -= 360.0;
            break;

        case ' X' :
            xrot -= 2.0;
            if (xrot < 0.0) xrot += 360.0;

```

```

        break;

    case ' y' :
        yrot += 2.0;
        if (yrot > 360.0) yrot -= 360.0;
        break;

    case ' Y' :
        yrot -= 2.0;
        if (yrot < 0.0) yrot += 360.0;
        break;

    case ' z' :
        zrot += 5.0;
        if (yrot > 360.0) zrot -= 360.0;
        break;

    case ' Z' :
        zrot -= 5.0;
        if (yrot < 0.0) zrot += 360.0;
        break;

    case ' d' :
        scaleFactor = scaleFactor*0.9;
        if (scaleFactor < 0.000005)
        {
            scaleFactor = 0.000005;
        }
        bbzSet= 0;
        FillNodeZCoords();
        ConstructVertexNormals();
        reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
        break;

    case ' D' :
        scaleFactor = scaleFactor*1.1;
        bbzSet= 0;
        FillNodeZCoords();
        ConstructVertexNormals();
        reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
        break;

    case ' e' :
        eigNum += 1;
        if (eigNum > numOfNodes) eigNum = numOfNodes;
        bbzSet= 0;
        FillNodeZCoords();
        ConstructVertexNormals();
        reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
        break;

    case ' E' :
        eigNum -= 1;
        if (eigNum < 0) eigNum = 0;

```

```

    bbzSet= 0;
    FillNodeZCoords();
    ConstructVertexNormals();
    reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
    break;

case ' f' :
    if (polyFillMode == GL_FILL)
    {
        polyFillMode = GL_LINE;
    }
    else
    {
        polyFillMode = GL_FILL;
    }
    break;

case ' r' :
    SetDefaults();
    reshape(glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));
    break;

case 27:
    exit(0);
    break;

default:
    break;
}

glutPostRedisplay();
}

// crossProd = vect1 X vect2
void CrossProduct(const aVector vect1, const aVector vect2, aVector& crossProd)
{
    crossProd[0] = vect1[1] * vect2[2] - vect1[2] * vect2[1];
    crossProd[1] = vect1[2] * vect2[0] - vect1[0] * vect2[2];
    crossProd[2] = vect1[0] * vect2[1] - vect1[1] * vect2[0];
}

void Normalize(aVector& vec)
{
    GLdouble length = sqrt(vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2]);

    vec[0] = vec[0]/length;
    vec[1] = vec[1]/length;
    vec[2] = vec[2]/length;
}

// vec = vect1 - vect2
void VectorSubtraction(const aVector vect1, const aVector vect2, aVector& vec)
{
    vec[0] = vect1[0]-vect2[0];

```

```

    vec[1] = vect1[1]-vect2[1];
    vec[2] = vect1[2]-vect2[2];
}

// vec = vect1 + vect2
void VectorAddition(const aVector vect1, const aVector vect2, aVector& vec)
{
    vec[0] = vect1[0]+vect2[0];
    vec[1] = vect1[1]+vect2[1];
    vec[2] = vect1[2]+vect2[2];
}

void ConstructVertexNormals(void)
{
    aVector partialNormal = {0};

    for (int iTriangle = 0; iTriangle < numOfTriangles; iTriangle++)
    {
        aVector leftArm = {0};
        aVector rightArm = {0};

        VectorSubtraction(nodeCoords[triangles[iTriangle][0]],
                        nodeCoords[triangles[iTriangle][1]],
                        leftArm);
        VectorSubtraction(nodeCoords[triangles[iTriangle][2]],
                        nodeCoords[triangles[iTriangle][1]],
                        rightArm);

        CrossProduct(rightArm, leftArm, partialNormal);
        Normalize(partialNormal);

        triNorms[iTriangle][0] = partialNormal[0];
        triNorms[iTriangle][1] = partialNormal[1];
        triNorms[iTriangle][2] = partialNormal[2];

        VectorAddition(vertNorms[triangles[iTriangle][0]], partialNormal,
                        vertNorms[triangles[iTriangle][0]]);
        // add partial normal to the vertex at the right angle twice
        // 90 degree angle twice as large as 45 degrees
        VectorAddition(vertNorms[triangles[iTriangle][1]], partialNormal,
                        vertNorms[triangles[iTriangle][1]]);
        VectorAddition(vertNorms[triangles[iTriangle][1]], partialNormal,
                        vertNorms[triangles[iTriangle][1]]);
        VectorAddition(vertNorms[triangles[iTriangle][2]], partialNormal,
                        vertNorms[triangles[iTriangle][2]]);
    }

    for (int iVert = 0; iVert < numOfNodes; iVert++)
    {
        Normalize(vertNorms[iVert]);
    }
}

void SetDefaults()

```

```

{
    xrot = 0.0;
    yrot = 0.0;
    zoomFactor = 1.0;
    scaleFactor = 1.0;
    bbzSet = 0;
    polyFillMode = GL_FILL;
    eigNum = 0;

    FillNodeZCoords();
}

void DisplayMain(int argc, char* argv[])
{
    SetDefaults();

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    makeMenu();
    glutSpecialFunc(specialKey);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(key);
    glutMainLoop();
}

// *****
// FEM2DLinear.h
//
// Finite Element Method with Two-Dimensional Linear Triangles
//
// Eigensolution implementation, header file
//
// Jason C. Hunnell
//
// March 8, 2002
//
// *****

#ifndef FEM2DLINEAR_H
#define FEM2DLINEAR_H

#include <cmath>
#include <iomanip>
#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>
#include "TNT\tnt.h"
#include "TNT\vec.h"
#include "TNT\cmat.h"

```

```

#include "TNT\fmt.h"
#include "TNT\transv.h"

using namespace std;
using namespace TNT;

typedef GLdouble aVector[3];

struct QuadPolyCoeffs
{
    //  $c + xX + yY + xxX^2 + xyX*Y + yyY^2$ 
    double c;
    double x;
    double y;
    double xx;
    double xy;
    double yy;
};

void AssembleLinearElement(int triangleNumber, Matrix<double> &K, Matrix<double> &M);
double AlphaX(const int &triangleNumber, const double &x, const double &y);
double AlphaY(const int &triangleNumber, const double &x, const double &y);
double Beta(const int &triangleNumber, const double &x, const double &y);
double Gamma(const int &triangleNumber, const double &x, const double &y);
void SolveGenSymEigen(const Matrix<double> &A, const Matrix<double> &B,
    Matrix<double> &Eigenvectors, Vector<double> &Eigenvalues,
    int &Info);
Matrix<double> K;
Matrix<double> M;
Matrix<double> Eigenvecs;
Vector<double> Eigenvals;
int Lx; // linear dimension in x
int Ly; // linear dimension in y
int Ix; // number of grid partitions in x
int Iy; // number of grid partitions in y
int numOfNodes; // number of nodes
int numOfTriangles; // number of triangles
int eigNum; // the eigenvector to display

double (*nodeCoords)[3]; // array of node coordinates
int (*triangles)[3]; // array of triangle nodes and attribute indices
double (*vertNorms)[3]; // array of vertex normal vectors
double (*triNorms)[3]; // array of triangle normal vectors

// the attributes, {alphaXCoeffs,alphaYCoeffs,betaCoeffs,gammaCoeffs}
QuadPolyCoeffs attributes[4] = {0};

GLfloat xrot; // x rotation angle
GLfloat yrot; // y rotation angle
GLfloat zrot; // z rotation angle
GLfloat radius; // bounding radius
GLfloat zoomFactor; // zoom factor
GLfloat scaleFactor; // scale factor

```



```

GLenum polyFillMode; // polygon fill mode
GLfloat bbzmin = 0.0; // bounding box zmin
GLfloat bbzmax = 0.0; // bounding box zmax
int bbzSet = 0; // bounding box zmin and zmax have been set

void SetModelData();
void Solve();
void CleanUp();
void OutputSolution(ofstream&);
void FillNodeCoords(void);
void FillNodeZCoords(void);
void FillTriangles(void);
void ConstructVertexNormals(void);
void CrossProduct(const aVector vect1, const aVector vect2, aVector& crossProd);
void Normalize(aVector& vec);
void VectorSubtraction(const aVector vect1, const aVector vect2, aVector& vec);
void VectorAddition(const aVector vect1, const aVector vect2, aVector& vec);
void SetDefaults(void);
void DisplayMain(int argc, char* argv[]);
void display(void);
void init(void);
void specialKey(int key, int x, int y);
void key(unsigned char key, int x, int y);
void makeMenu(void);
void menu(int item);
void reshape(int w, int h);

extern "C"
{
// solve general symmetric eigenvalues, eigenvectors
//
void dsygv_( const int *itype, char *jobz, char *uplo, const int *N,
             double *A, const int *lda, double *B, const int *ldb, double *W,
             double *work, const int *lwork, int *info);
}

#endif
// End of header file FEM2DLinear.h

```

## REFERENCES

1. K.J. Bathe, *Finite Element Procedures* (Prentice-Hall, New Jersey, 1996); R. D.Cook, D. S. Malkus, and M. E. Plesha, *Concepts and Applications of Finite Element Analysis* (John Wiley & Sons, New York, 1989), 3<sup>rd</sup> ed.; J. N. Reddy, *An Introduction to the Finite Element Method* (McGraw-Hill, New York, 1993), 2<sup>nd</sup> ed.
2. D. S. Burnett, *Finite Element Analysis From Concepts to Applications* (Addison-Wesley, Reading, Mass., 1987).
3. L. R. Ram-Mohan, S. Saigal, D. Dossa, and J. Shertzer, *Comput. in Physics* **4**, 59 (1990).
4. R. Goloskie, J. W. Kramer, and L. R. Ram-Mohan, *Computers in Physics* **8**, 679 (1994).
5. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide* (SIAM Publications, Philadelphia, 1999), 3<sup>rd</sup> ed.
6. Mathematical and Computational Sciences Division, National Institute of Standards and Technology, Gaithersburg, MD USA, *Template Numerical Toolkit (TNT): Linear Algebra Module* (<http://math.nist.gov/tnt>, 2000).